Modelos de Computación I

Serafín Moral
Departamento de Ciencias de la Computación e I.A.
ETSI Informática
Universidad de Granada

Índice general

Capítulo 1

Introducción

Hoy en día parece que no existe ningún límite a lo que un ordenador puede llegar a hacer, y da la impresión de que cada vez se pueden resolver nuevos y más difíciles problemas.

Casi desde que aparece sobre La Tierra, el hombre ha tratado de buscar procedimientos y máquinas que le facilitasen la realización de cálculos (aritméticos primero, y otros más complejos posteriormente).

El avance tecnológico para representar datos y/o información por un lado, y el diseño de nuevas formas de manejarlos, propicia el desarrollo de dispositivos y máquinas de calcular.

Un aspecto importante en el desarrollo de los ordenadores, es sin duda, su aplicación para resolver problemas científicos y empresariales. Esta aplicación hubiese resultado muy difícil sin la utilización de procedimientos que permiten resolver estos problemas mediante una sucesión de pasos claros, concretos y sencillos, es decir algorítmos. El avance de las matemáticas permite la utilización de nuevas metodologías para la representación y manejo de la información.

Por otro lado, aparece el intento de los matemáticos y científicos para obtener un procedimiento general para poder resolver cualquier problema (matemático) claramente formulado. Es lo que podríamos llamar **El problema de la computación teórica**. El avance de la tecnología y de las matemáticas, y más en concreto de la teoría de conjuntos y de la lógica, permiten plantearse aspectos de la computación en 3 caminos.

a) Computación teórica. Autómatas, Funciones Recursivas, ...

b) Ordenadores digitales. Nuevas tecnologías, nuevos lenguajes, ...

c) Intentos de modelizar el cerebro biológico

   1. Redes Neuronales (intentan modelizar el "procesador")
   2. Conjuntos y Lógica Difusa (representar y manejar la información)

En este texto veremos aspectos relativos a a) y c.1).

Considermos el problema general, bajo un planteamiento de Programación.

Desde un punto de vista global, un programa (traducido a lenguaje máquina) se puede ver como una sucesión de ceros y unos, cuya ejecución produce una salida, que es otra serie de ceros y unos. Si añadimos un 1 al inicio de cada cadena binaria (programa y salida), podemos entender los programas como aplicaciones concretas de una función entre números naturales en binario. El argumento (variable independiente) sería el programa y la función (var. dependiente) la salida del programa.

Obviamente, el número de funciones posibles de N en N es **nonumerable**, mientras que el número de posibles programas que podemos escribir con un Lenguaje de Programación, que tiene un número finito de símbolos, es **numerable**.

Esto significa (cada programa puede calcular una sola función como las indicadas antes) que existen muchas funciones para las que no podemos escribir un programa en nuestro L. de Alto Nivel, aunque seguramente estamos interesados en resolver el problema asociado a la función.

Entonces nos preguntamos cosas como:

¿Para qué problemas no podemos escribir un programa ?

¿Podremos resolver algunos de estos problemas con otro lenguaje de programación y/o con otros computadores ?.

Además, para los problemas que si tienen un programa asociado,

¿Se podrá ejecutar el programa en un ordenador actual en un tiempo razonable ? (p.e., antes de que llegue nuestra jubilación).

¿Los ordenadores futuros podrán hacerlo ?

Trataremos de dar respuestas a algunas de estas cuestiones, a lo largo del desarrollo de la asignatura.

1.1. Introducción Histórica

Uno de los principales factores determinantes de la profunda revolución experimentada en el ámbito de la ciencia, la técnica y la cultura de nuestros días es el desarrollo de la informática. La palabra 'informática' (**Infor**mación auto**mática**), es un nombre colectivo que designa un vasto conjunto de teorías y técnicas científicas -desde la matemática abstracta hasta la ingeniería y la gestión administrativa- cuyo objeto es el diseño y el uso de los ordenadores. Pero el núcleo teórico más sólido y fundamental de todo ese conjunto de doctrinas y prácticas es la llamada 'Teoría de la Computabilidad', formalmente elaborada en los años 30 y 40 gracias a los descubrimientos de lógicos matemáticos como Gödel, Turing, Post, Church, y Kleene, aunque sus orígenes más remotos datan de antiguo, con el planteamiento de la cuestión de saber si, al cabo de cierto esfuerzo, el hombre podría llegar a un extremo en la investigación en que, eventualmente, toda clase de problemas pudiera ser atacado por un procedimiento general de forma que no requiriera el más leve esfuerzo de imaginación creadora para llevarlo a cabo. Si todo queda determinado así en detalle, entonces sería obviamente posible abandonar la ejecución del método a una máquina, máxime si la máquina en cuestión es totalmente automática. Esta ídea, ambiciosa sin duda, ha influido poderosamente en diferentes épocas el desarrollo de la ciencia.

El propósito inicial es hacer precisa la noción intuitiva de función calculable; esto es, una función cuyos valores pueden ser calculados de forma automática o efectiva mediante un algoritmo, y construir modelos teóricos para ello (de computación). Así podemos obtener una comprensión más clara de esta idea intuitiva; y solo de esta forma podemos explorar matemáticamente el concepto de computabilidad y los conceptos relacionadas con ella, tales como decibilidad, etc...

La teoría de la computabilidad puede caracterizarse, desde el punto de vista de las C.C., como la búsqueda de respuestas para las siguientes preguntas:

1)¿Qué pueden hacer los ordenadores (sin restricciones de ningún tipo)?

2) ¿Cuales son las limitaciones inherentes a los métodos automáticos de cálculo?.

El primer paso en la búsqueda de las respuestas a estas preguntas está en el estudio de los modelos de computación.

Los comienzos de la Teoría. La Tesis de Church-Turing

Los modelos abstractos de computación tienen su origen en los años 30, bastante antes de que existieran los ordenadores modernos, en el trabajo de los lógicos Church, Gödel, Kleene, Post, y Turing. Estos primeros trabajos han tenido una profunda influencia no solo en el desarrollo teórico de las C.C., sino que muchos aspectos de la práctica de la computación que son ahora lugar común de los informáticos, fueron presagiados por ellos; incluyendo la existencia de ordenadores de propósito general, la posibilidad de interpretar programas, la dualidad entre software y hardware, y la representación de lenguajes por estructuras formales basados en reglas de producción.

El punto de partida de estos primeros trabajos fueron las cuestiones fundamentales que D. Hilbert formuló en 1928, durante el transcurso de un congreso internacional:
1.- ¿Son completas las matemáticas, en el sentido de que pueda probarse o no cada aseveración matemática?
2.- ¿Son las matemáticas consistentes, en el sentido de que no pueda probarse simultaneamente una aseveración y su negación ?
3.- ¿Son las matemáticas decidibles, en el sentido de que exista un método definido que se pueda aplicar a cualquier aseveración matemática, y que determine si dicha aseveración es cierta?.

La meta de Hilbert era crear un sistema matemático formal çompletoz çonsistente", en el que todas las aseveraciones pudieran plantearse con precisión. Su idea era encontrar un algoritmo que determinara la verdad o falsedad de cualquier proposición en el sistema formal. A este problema le llamó el 'Entscheidungsproblem'.

Por desgracia para Hilbert, en la década de 1930 se produjeron una serie de investigaciones que mostraron que esto no era posible. Las primeras noticias en contra surgen en 1931 con K. Gödel y su Teorema de Incompletitud: "Todo sistema de primer orden consistente que contenga los teoremas de la aritmética y cuyo conjunto de (números de Gödel de) axiomas sea recursivo no es completo."

Como consecuencia no será posible encontrar el sistema formal deseado por Hilbert en el marco de la lógica de primer orden, a no ser que se tome un conjunto no recursivo de axiomas, hecho que escapaba a la mente de los matemáticos. Una versión posterior y más general del teorema de Gödel elimina la posibilidad de considerar sistemas deductivos más potentes que los sistemas de primer orden, demostrando que no pueden ser consistentes y completos a la vez.

Un aspecto a destacar dentro del teorema de incompletitud de Gödel, fué la idea de codificación. Se indica un método (numeración de Gödel) mediante el cual se asigna un número de código (entero positivo) a cada fórmula bien formada del sistema (fbf) y a cada sucesión finita de fórmulas bien formadas, de tal modo que la fbf o sucesión finita de fbf se recupera fácilmente a partir de su número de código. A través de este código, los enunciados referentes a enteros positivos, pueden considerarse como enunciados referentes a números de código de expresiones, o

incluso referentes a las propias expresiones. Esta misma idea fué posteriormente utilizada para codificar algoritmos como enteros positivos, y así poder considerar un algoritmo, cuyas entradas fuesen enteros positivos, como un algoritmo cuyas entradas fuesen algoritmos.

El siguiente paso importante lo constituye la aparición casi simultanea en 1936 de varias caracterizaciones independientes de la noción de calculabilidad efectiva, en los trabajos de Church, Kleene, Turing y Post. Los tres primeros mostraban problemas que eran efectivamente indecidibles; Church y Turing probaron además que el Entscheidungsproblem era un problema indecidible.

Church propuso la noción de función λ-definible como función efectivamente calculable. La demostración de teoremas se convierte en una transformación de una cadena de símbolos en otra, en cálculo lambda, según un conjunto de reglas formales. Este sistema resultó ser inconsistente, pero la capacidad para expresar-calcular funciones numéricas como términos del sistema llamó pronto la atención de él y sus colaboradores.

Gödel habia recogido la idea de Herbrand de que una función f podría definirse por un conjunto de ecuaciones entre términos que incluian a la función f y a símbolos para funciones previamente definidas, y precisó esta idea requiriendo que cada valor de f se obtenga de las ecuaciones por sustitución de las variables por números y los términos libres de variables por los valores que ya se habian probado que designaban. Esto define la clase de 'las funciones recursivas de Herbrand-Gödel'.

En 1936, Church hace un esquema de la demostración de la equivalencia entre las funciones λ-definibles y las funciones recursivas de Herbrand-Gödel (esta equivalencia también había sido probada por Kleene); y aventura que estas iban a ser las únicas funciones calculables por medio de un algoritmo a través de la tesis que lleva su nombre, y utilizando la noción de función λ-definible, dió ejemplos de problemas de decisión irresolubles, y demostró que el Entscheidungsproblem era uno de esos problemas.

Por otra parte Kleene, pocos meses despues, demuestra formalmente la equivalencia entre funciones λ-definible y funciones recursivas de Herbrand-Gödel, y dá ejemplos de problemas irresolubles utilizando la noción de función recursiva.

La tercera noción de función calculable proviene del matemático inglés A. Turing, quién argumentó que la tercera cuestión de Hilbert (el Entscheidungsproblem) podía atacarse con la ayuda de una máquina, al menos con el concepto abstracto de máquina.

Turing señaló que había tenido éxito en caracterizar de un modo matemáticamente preciso, por medio de sus máquinas, la clase de las funciones calculables mediante un algoritmo, lo que se conoce hoy como *Tesis de Turing*.

Aunque no se puede dar ninguna prueba formal de que una máquina pueda tener esa propiedad, Turing dió un elevado número de argumentos a su favor, en base a lo cual presentó la tesis como un teorema demostrado. Además, utilizó su concepto de máquina para demostrar que existen funciones que no son calculables por un método definido y en particular, que el Entscheidungsproblem era uno de esos problemas.

Cuando Turing conoció los trabajos de Church-Kleene, demostró que los conceptos de función λ-definible y función calculable por medio de una máquina de Turing coinciden. Naturalmente a la luz de esto la Tesis de Turing resulta ser equivalente a la de Church.

Finalmente, cabe reseñar el trabajo de E. Post. Este estaba interesado en marcar la frontera entre lo que se puede hacer en matemáticas simplemente por procedimientos formales y lo que depende de la comprensión y el entendimiento. De esta forma, Post formula un modelo de procedimiento efectivo a través de los llamados sistemas deductivos normales. Estos son sistemas puramente formales en los que puede 'deducirse' sucesiones finitas de símbolos como consecuencia de otras sucesiones finitas de símbolos por medio de un tipo normalizado de reglas y a partir de un conjunto de axiomas. Así pues, dada una sucesión finita de símbolos como entrada, las reglas permiten convertirla en una sucesión finita de salida. En su artículo, Post demostró resultados de incompletitud e indecibilidad en estos sistemas.

Los resultados hasta ahora citados, se refieren a funciones totales. La existencia de algoritmos que con determinadas entradas nunca terminan, condujo de forma natural a considerar funciones parciales. Kleene fué el primero en hacer tal consideración en 1938. El estudio de estas funciones ha mostrado la posibilidad de generalizar todos los resultados anteriores a funciones parciales. Por otro lado, el estudio de las funciones parciales calculables ha resultado esencial para el posterior desarrollo de la materia.

Posteriormente, se demostró la equivalencia entre lo que se podía calcular mediante una máquina de Turing y lo que se podía calcular mediante un sistema formal en general.

A la vista de estos resultados, la Tesis de Church-Turing es aceptada como un axioma en la teoría de la computación, y ha servido como punto de partida en la investigación de los problemas que se pueden resolver mediante un algoritmo.

Problemas no computables

Usando la codificación de Gödel, se demostró que era posible construir una máquina de propósito general, es decir, capaz de resolver cualquier problema que se pudiese resolver mediante un algoritmo. Dicha máquina tendría como entrada el entero que codificaría el algoritmo solución del problema y la propia entrada del problema, de tal forma, que la máquina aplicaría el algoritmo codificado a la entrada del problema. Esta hipotética máquina puede considerarse como el padre de los actuales ordenadores de proposito general.

Una de las cuestiones más estudiadas en la teoría de la computabilidad ha sido la posibilidad de construir algoritmos que nos determinen si un determinado algoritmo posee o no una determinada propiedad. Así, sería interesante responder de forma automática a cuestiones como:
- ¿Calculan los algoritmos A y B la misma función? (Problema de la equivalencia)
- ¿Parará el algoritmo A para una de sus entradas? (Problema de la parada)
- ¿Parará el algoritmo A para todas sus entradas? (Problema de la totalidad)
- ¿Calcula el algoritmo A la función f? (Problema de la verificación?)

etc ...

En un principio se fueron obteniendo demostraciones individuales de la no computabilidad

de cada una de estas cuestiones, de forma que se tenía la sensación de que casi cualquier pregunta interesante acerca de algoritmos era no computable.

A pesar de esto, y como consecuencia de la existencia de un programa universal hay otras muchas cuestiones interesantes que se han demostrado computables.

El identificar los problemas que son computables y los que no lo son tiene un considerable interés, pues indica el alcance y los límites de la computabilidad, y así demuestra los límites teóricos de los ordenadores. Además de las cuestiones sobre algoritmos, se han encontrado numerosos problemas menos "generales"que han resultado ser no computables. Como ejemplo citamos:

- Décimo problema de Hilbert. Una ecuación diofántica es la ecuación de los ceros enteros de un polinomio con coeficientes enteros. Se pregunta si hay un procedimiento efectivo que determine si una ecuación diofántica tiene o no solución.

Por otro lado, son muchos los problemas interesantes que se han demostrado computables. Todas las funciones construidas por recursividad primitiva o minimalización a partir de funciones calculables resultan ser calculables como consecuencia de los trabajos de Church y Turing. Pero además, otras funciones más complejamente definidas también son computables. Como ejemplo más interesante de aplicación de este tipo de recursión tenemos la función de Ackermann ψ:

$$\psi(0,y) = y+1;$$

$$\psi(x+1,0) = \psi(x,1);$$

$$\psi(x+1,y+1) = \psi(x,\psi(x+1,y)).$$

1.2. Diferentes Modelos de Computación

Consideraremos las Ciencias de la Computación como un cuerpo de conocimiento cuyo principal objetivo es la resolución de problemas por medio de un ordenador. Podemos citar las siguientes definiciones:

a) La ACM (Asociation Computing Machinering):

‘la disciplina Ciencias de la Computación es el estudio sistemático de los procesos algorítmicos que describen y transforman información: teoría, análisis, diseño, eficiencia, implementación, y aplicación.’

b) Norman E. Gibbs y Allen B. Tucker (1986)indican que: ‘no debemos entender que el objetivo de las Ciencias de la Computación sea la construcción de programas sino el estudio sistemático de los algoritmos y estructura de datos, específicamente de sus propiedades formales’.

Para ser más concretos (A. Berztiss 1987), vamos a considerar a las C.C. como un cuerpo de conocimiento cuyo objetivo es obtener respuestas para las siguientes cuestiones:

A) ¿Qué problemas se pueden resolver mediante un ordenador?.

B) ¿Cómo puede construirse un programa para resolver un problema?.

C) ¿Resuelve realmente nuestro programa el problema?.

D) ¿Cuanto tiempo y espacio consume nuestro problema?.

Analizando en profundidad los 4 puntos anteriores llegaremos a descubrir explícitamente los diferentes contenidos abarcados por las C.C.

El planteamiento de la primera cuestión nos conduce a precisar el concepto de problema y de lo que un ordenador es capaz de realizar.

Durante muchos años se creyó que si un problema podía enunciarse de manera precisa, entonces con suficiente esfuerzo y tiempo sería posible encontrar un 'algoritmo' o método para encontrar una solución (o tal vez podría proporcionarse una prueba de que tal solución no existe). En otras palabras, se creía que no había problema que fuera tan intrínsecamente difícil que en principio nunca pudiera resolverse.

Uno de los grandes promotores de esta creencia fué el matemático David Hilbert (1862-1943), quien en un congreso mundial afirmó:

"Todo problema matemático bien definido debe ser necesariamente susceptible de un planteamiento exacto, ya sea en forma de una respuesta real a la pregunta planteada o debido a la constatación de la imposibilidad de resolverlo, a lo que se debería el necesario fallo de todos los intentos... "

El principal obstáculo que los matemáticos de principios de siglo encontraban al plantearse estas cuestiones era concretar con exactitud lo que significa la palabra algoritmo como sinónimo de método para encontrar una solución. La noción de algoritmo era intuitiva y no matemáticamente precisa.

Las descripciones dadas por los primeros investigadores tomaron diferentes formas, que pueden clasificarse ampliamente del siguiente modo:

(a) máquinas computadoras abstractas (definidas de modo preciso),

(b) construcciones formales de procedimientos de cómputo, y

(c) construcciones formales productoras de clases de funciones.

Las dos primeras caracterizaciones se refieren a la propia noción de algoritmo (en principio no hay gran diferencia entre ambas). La última dá descripciones de la clase de funciones computables mediante un algoritmo.

Ejemplos de (a) son los Autómatas y las *máquinas de Turing*, (diseñadas por Turing en los años 30). Un ejemplo de (b) son los *sistemas de Thue*. Por último, las *funciones recursivas* constituyen el ejemplo clásico de (c).

El resultado crucial es que las diversas caracterizaciones de las funciones (parciales) computables mediante un algoritmo condujeron todas a una misma clase, a saber, la clase de las funciones parciales recursivas. Esto es algo suceptible de demostración, y que ha sido demostrado. Lo que no es suceptible de demostración es que la clase de las funciones parciales recursivas coincida con la clase de las funciones computables mediante un algoritmo. No obstante, a la luz

de las evidencias a favor y de la falta de evidencias en contra, aceptamos la *Tesis de Church* que afirma la equivalencia de ambas clases.

Se clasifican los problemas según que siempre sea posible encontrar la solución por medio de un algoritmo (problemas computables) ó que no existan algoritmos que siempre produzcan una solución (problemas no computables).

Surge de modo inmediato la cuestión B) de como diseñar un programa (algoritmo especificado para poder ser ejecutado por un ordenador) que resuelva un problema dado. En la primera época del desarrollo informático los programas dependían intrínsecamente del ordenador utilizado, pues se expresaban en lenguaje máquina, directamente interpretable por el ordenador.

Surgió entonces la necesidad de idear otros mecanismos para construir y expresar los programas. El hilo conductor de tales mecanismos fué la abstracción: separar el programa del ordenador y acercarlo cada vez más al problema.

Los subprogramas empezaron ya a usarse a principios de los 50, dando lugar posteriormente al primer tipo de abstracción, la procedimental. A principios de los 60, se empezaron a entender los conceptos abstractos asociados a estructuras de datos básicas pero aún no se separaban los conceptos de las implementaciones. Con el nacimiento en esta época de los primeros lenguajes de alto nivel, Fortran p.ej., se llegó a la abstracción sintáctica, al abstraerse la semántica de las expresiones matemáticas y encapsular el acceso a ellas a través de la sintaxis propia del lenguaje. En cualquier caso con el desarrollo de estos lenguajes de alto nivel se solventaron los problemas de flexibilidad en la comunicación con el ordenador, y se empezaron a estudiar los algoritmos de forma independiente del ordenador concreto en que se probaran y del lenguaje concreto en que se expresaran.

Aparece la necesidad de traducir los programas escritos en lenguajes de alto nivel al lenguaje máquina, de forma automática, y se buscan máquinas o procedimientos que puedan reconer el léxico y la sintáxis de dichos lenguajes.

Hay que comentar que no hay un algoritmo para enseñar a diseñar algoritmos, y que muchas veces el proceso de construcción puede llegar a ser muy poco disciplinado. No obstante, existen técnicas de diseño de algoritmos, que vienen a ser modelos abstractos de los mismos aplicables a gran variedad de problemas reales.

Una vez construido un programa para un problema, surge la cuestión C) de si lo resuelve realmente. Normalmente los programadores prueban sus programas sobre una gran cantidad de datos de entrada para descubrir la mayoria de los errores lógicos presentes, aunque con este método (al que suele denominarse de prueba y depuración) no se puede estar completamente seguro de que el programa no contiene errores. Necesitaríamos para realizar la verificación formal, reglas que describan de forma precisa el efecto que cada instrucción tiene en el estado actual del programa, para, aplicando dichas reglas demostrar rigurosamente que lo que hace el programa coincide con sus especificaciones. En cualquier caso y cuando la prueba formal resulte muy complicada, podemos aumentar la confianza en nuestro programa realizando en el mismo los "testçuidadosos de que hablábamos al principio.

Alcanzado este punto, ya tenemos un programa que en principio es solución de un problema. Se plantea entonces la duda de que hacer en caso de que para el mismo problema seamos capaces de construir otro programa que también lo resuelva. ¿Cómo decidirnos por una u otra solución? o más aún, ¿qué ocurre si el programa aún siendo correcto consume demasiados recursos y es inaceptable?. La respuesta viene dada a través del punto D) en nuestro recorrido: el análisis del tiempo y espacio que necesita una solución concreta; en definitiva, el estudio de la eficiencia de los programas, midiendo la complejidad en espacio, por el número de variables y el número y tamaño de las estructuras de datos que se usan, y la complejidad en tiempo por el número de acciones elementales llevadas a cabo en la ejecución del programa.

Los problemas computables fueron entonces clasificados en dos tipos: problemas eficientemente computables, para los que existía un algoritmo eficiente; y problemas intratables, para los que no existen algoritmos eficientes. La existencia de problemas intratables no ha sido probada, si bién se han encontrado muchas evidencias a su favor.

Otra clase de problemas a considerar es la clase NP de los problemas para los que existia un algoritmo no determinístico en tiempo polinomial, y dentro de ella, los problemas NP-completos.

Los intentos (desde los años 40) de construir máquinas para modelizar algunas de las funciones del cerebro biológico, ha permitido desarrollar máquinas capaces de 'aprender' (y reproducir) funciones (o sistemas) cuya forma (o comportamiento) se desconoce, pero sí conocemos una serie de ejemplos que reflejan esta forma (o comportamiento). Estas máquinas llamadas Redes Neuronales Artificiales también aportan su granito de arena al desarrollo de la computación.

A menudo se utiliza la técnica de reducir un problema a otro para comprobar si tiene o no solución efectiva. La estratégia en el caso de la respuesta negativa es la siguiente, si se reduce de forma efectiva un problema sin solución efectiva a otro problema (mediante una función calculable), entonces este nuevo problema tampoco tendrá solución efectiva. La razón es muy simple, si tuviese solución efectiva, componiendo el algoritmo solución con el algoritmo de transformación obtendríamos una solución para el problema efectivamente irresoluble. En sentido inverso, si se reduce un problema a otro para el que se conoce una solución efectiva, entonces componiendo se obtiene una solución para el primer problema. Esta técnica es muy útil y se utiliza a menudo. Por otro lado, esta mísma técnica es muy empleada en el campo de la complejidad algorítmica.

La Complejidad Algorítmica trata de estudiar la relativa dificultad computacional de las funciones computables. Rabin (1960) fué de los primeros en plantear la cuestión ¿Qué quiere decir que f sea más difícil de computar que g?

J. Hartmanis and R.E. Stearns, en *On the computational complexity of algorithms* (1965) introducen la noción fundamental de medida de complejidad definida como el tiempo de computación sobre una máquina de Turing multicinta.

Después surge la definición de funciones computables en tiempo polinomial, y se establece una jerarquía de complejidad, los problemas NP, NP-duros y NP-completos. De todas formas, el

perfil de la plaza no se ocupa directamente de este problema que además se estudia ampliamente en otras materias y asignaturas del curriculum de Informática.

1.2.1. Autómatas y Lenguajes

El desarrollo de los ordenadores en la década de los 40, con la introducción de los programas en la memoria principal, y posteriormente con los lenguajes de programación de alto nivel, propician la distinción entre lenguajes formales, con reglas sintácticas y semánticas rígidas, concretas y bién definidas, de los lenguajes naturales como el inglés, donde la sintaxis y la semántica no se pueden controlar fácilmente. Los intentos de formalizar los lenguajes naturales, lleva a la construcción de gramáticas, como una forma de describir estos lenguajes, utilizando para ello reglas de producción para construir las frases del lenguaje. Se puede entonces caracterizar un Lenguaje, mediante las reglas de una gramática adecuada.

Los trabajos de McCulloch y Pitts (1943) describen los cálculos lógicos inmersos en un dispositivo (neurona artificial) que habían diseñado para simular la actividad de una neurona biológica. El dispositivo recibía o no, una serie de impulsos eléctricos por sus entradas que se ponderaban, y producía una salida binaria (existe pulso eléctrico o no). Las entradas y salidas se podían considerar como cadenas de 0 y 1, indicando entonces la forma de combinar la cadena de entrada para producir la salida. La notación utilizada es la base para el desarrollo de expresiones regulares en la descripción de conjuntos de cadenas de caracteres.

C. Shannon (1948) define los fundamentos de la teoría de la información, y utiliza esquemas para poder definir sistemas discretos, parecidos a los autómatas finitos, relacionándoloss con cadenas de Markov, para realizar aproximaciones a los lenguajes naturales.

J. Von Neumann (1948) introduce el t'ermino de teoría de autómatas, y dice sobre los trabajos de McCulloch-Pitts: *".. el resultado más importante de McCulloch-Pitts, es que cualquier funcionamiento en este sentido, que pueda ser definido en todo, lógicamente, estríctamente y sin ambiguedad, en un número finito de palabras, puede ser realizado también por una tal red neuronal formal"*

La necesidad de traducir los algorítmos escritos en lenguajes de alto nivel al lenguaje máquina, propicia la utilización de máquinas como los autómatas de estados finitos, para reconocer si una cadena determinada pertenece (es una frase de) a un lenguaje concreto, usando para ello la función de transición de estados, mediante un diagrama de transición o una tabla adecuada. Tenemos así otra forma de caracterizar los lenguajes, de acuerdo con máquinas automáticas que permitan reconocer sus frases.

S.C. Kleene, en 1951, realiza un informe (solicitado por la RAND Corporation) sobre los trabajos de McCulloch-Pitts, que se publica en 1956. En este informe, Kleene demuestra la equivalencia entre lo que él llama "dos formas de definir una misma cosa", que son los *sucesos regulares* (que se pueden describir a partir de sucesos bases y los operadores unión, concatenación e iteración (*)), es decir, expresiones regulares, y sucesos especificados por un autómata

finito.

Rabin y Scott (1960) obtienen un modelo de computador con una cantidad finita de memoria, al que llamaron autómata de estados finitos. Demostraron que su comportamiento posible, era básicamente el mismo que el descrito mediante expresiones regulares, desarrolladas a partir de los trabajos de McCulloch y Pitts.

No obstante lo dicho, para un alfabeto concreto, no todos los lenguajes que se pueden construir son regulares. Ni siquiera todos los interesantes desde el punto de vista de la construcción de algorítmos para resolver problemas. Hay entonces muchos problemas que no son calculables con estos lenguajes. Esto pone de manifiesto las limitaciones de los autómatas finitos y las gramáticas regulares, y propicia el desarrollo de máquinas reconocedoras de otros tipos de lenguajes y de las gramáticas correspondientes, asociadas a los mismos.

En 1956, la Princenton Univ. Press publica el libro *Automata Studies*, editado por C. Shannon y J. McCarthy, donde se recogen una serie de trabajos sobre autómatas y lenguajes formales. D. A. Huffman (1954) ya utiliza conceptos como *estado de un autómata y tabla de transiciones*.

N. Chomsky (1956) propone tres modelos para la descripción de lenguajes, que son la base de su futura jerarquía de los tipos de lenguajes, que ayudó también en el desarrollo de los lenguajes de programación. Para ello intentó utilizar autómatas para extraer estructuras sintácticas (*"... el inglés no es un lenguaje de estados finitos"*) y dirige sus estudios a las gramáticas, indicando que la diferencia esencial entre autómatas y gramáticas es que la lógica asociada a los autómatas (p.e., para ver la equivalencia entre dos de ellos) es Decidible, mientras que la asociada a las gramáticas no lo es.

Desarrolla el concepto de gramática libre del contexto, en el transcurso de sus investigaciones sobre la sintáxis de los lenguajes naturales.

Backus y Naur desarrollaron una notación formal para describir la sintáxis de algunos lenguajes de programación, que básicamente se sigue utilizando todavía, y que podía considerarse equivalente a las gramáticas libres del contexto.

Consideramos entonces los lenguajes libres (independientes) del contexto, y las gramáticas libres del contexto y los autómatas con pila, como forma de caracterizarlos y manejarlos. Los distintos lenguajes formales que se pueden construir sobre un alfabeto concreto pueden clasificarse en clases cada vez más amplias que incluyen como subconjunto a las anteriores, de acuerdo con la jerarquía establecida por Chomsky en los años 50.

Se puede llegar así, de una forma casi natural a considerar las máquinas de Turing, establecidas casi 20 años antes, como máquinas reconocedoras de los lenguajes formales dependientes del contexto o estructurados por frases, e incluso a interpretar la Tesis de Turing como que ün sistema computacional nunca podrá efectuar un análisis sintáctico de aquellos lenguajes que están por encima de los lenguajes estructurados por frases, según la jerarquía de Chomsky".

En consecuencia, podemos utilizar la teoría de autómatas y los conceptos relativos a gramáticas sobre distintos tipos de lenguajes, para decidir (si se puede) si una función (o problema) es calculable, en base a que podamos construir un algoritmo solución mediante un lenguaje que

puede ser analizado mediante alguna máquina de las citadas anteriormente.

Los temas sobre autómatas, computabilidad, e incluso la complejidad algorítmica fueron incorporándose a los curriculum de ciencias de la computación de diferentes universidades, mediada la década de los 60. Esta incorporación puso de manifiesto que las ciencias de la computación habían usado gran cantidad de ideas de muy diferentes campos para su desarrollo, y que la investigación sobre aspectos básicos podía cooperar y aumentar los avances de la computación.

1.3. Lenguajes y Gramáticas. Aspectos de su traducción

Una idea básica en programación es la enorme diferencia entre los lenguajes naturales (LN) y los lenguajes de programación (LP), fundamentalmente porque los LP tienen unas reglas de sintáxis y de semántica mucho más rígidas, lo que les hace manejables en los computadores.

En los LN, las reglas gramaticales se desarrollan para reglamentar"de alguna forma la propia evolución del lenguaje. Se trata pues de "explicar"la estructura del lenguaje, y no delimitarla. Esto obliga a reglas muy complejas y que se quedan obsoletas rápidamente.

Los lenguajes más formalizados (LF) como los lengujaes de progrmación o el lenguaje matemático, tienen unas estructuras claramente definidas y determinadas por sus reglas gramaticales (sintácticas y semánticas). Esto ha posibilitado y propiciado la construcción de traductores automáticos para estos lenguajes.

En el proceso de traducción que realizan los compiladores, la primera tarea que se realiza, es la identificación de *tokens* como bloques u objetos individuales contenidos en el "diccionario"del lenguaje (p.e., identificadores, palabras claves, operadores, ...). Esto lo realiza un módulo del compilador que es el **analizador de léxico**, que transforma el programa fuente, en una secuencia de tokens representando cada uno un solo objeto.

El siguiente paso, realizado por el **analizador sintáctico**, es identificar los tokens que forman parte de cada instrucción y ver que estas estén correctamente escritas. El tercer paso es generar el código con el **generador de código**.

Los dos primeros pasos requieren imperiosamente unas reglas gramaticalesçlaramente definidas, en las que apoyarse para la automatización del proceso de traducción. Generalmente se utilizan en estas tares, máquinas (o algoritmos) como los autómatas, que estudiaremos más adelante.

Vamos a precisar previamente, los conceptos básicos relativos a Lenguajes y Gramáticas formales.

1.3.1. Alfabetos y Palabras

Los dispositivos que vamos a estudiar trabajan con símbolos y cadenas (o palabras) fundamentalmente. En este apartado vamos a definir de forma precisa estos conceptos.

Definición 1 *Un alfabeto es un conjunto finito A. Sus elementos se llamarán símbolos o letras.*

Para notar los alfabetos, en general, usaremos siempre que sea posible las primeras letras en mayúsculas: $A,B,C,\ldots$. Para los símbolos trataremos de emplear las primeras letras en minúsculas: $a,b,c,\ldots$ o números.

Las siguientes son algunas normas de notación que respetaremos en lo posible a lo largo del curso. En casos particulares en los que sea imposible seguirlas lo indicaremos explicitamente.

Ejemplo 1 $A=\{0,1\}$ *es un alfabeto con símbolos* 0 *y* 1.

También es un alfabeto $B=\{<0,0>,<0,1>,<1,0>,<1,1>\}$ *con símbolos* $<0,0>$ $,<0,1>,<1,0>$ *y* $<1,1>$. *En este caso no hay que confundir los símbolos del alfabeto B con los síimbolos del lenguaje (o más precisamente meta-lenguaje) que usamos para expresarnos todos los días. Son dos cosas totalmente distintas. Nosotros para comunicarnos usamos un lenguaje que tiene unos símbolos que son las letras del alfabeto latino, los números, las letras del alfabeto griego, y una serie de símbolos especiales propios del lenguaje matemático (el meta-lenguaje). Con este lenguaje también definimos los conceptos de alfabeto y símbolo. Ahora bién, los símbolos de un lenguaje no tienen que ser algunos de los usados en el meta-lenguaje, sino que cada uno puede estar formado por 0, 1 o más símbolos del metalenguaje. De hecho como en este último caso (lenguaje B), un símbolo del alfabeto definido está formado por varios símbolos del metalenguaje. Para que esto no de lugar a confusión, siempre que ocurra una situación similar, encerraremos los símbolos entre ángulos* $<\ldots>$. *Esto indicará que todo lo que aparece es un único símbolo.*

Definición 2 *Una palabra sobre el alfabeto A es una sucesión finita de elementos de A. Es decir u es una palabra sobre A, si y solo si* $U=a_1\ldots a_n$ *donde* $a_i\in A$, $\forall i=1,\ldots,n$.

Por ejemplo, si $A=\{0,1\}$ entonces 0111 es una palabra sobre este alfabeto.

El conjunto de todas las palabras sobre un alfabeto A se nota como A^*.

Para las palabras usaremos, en lo posible, las últimas letras del alfabeto latino en minúsculas: $u,v,x,y,z,\ldots$.

Definición 3 *Si* $u\in A^*$, *entonces la logitud de la palabra u es el número de símbolos de A que contiene. La longitud de u se nota como* $|u|$. *Es decir si* $u=a_1\ldots a_n$, *entonce* $|u|=n$.

Definición 4 *La palabra vacía es la palabra de longitud cero. Es la misma para todos los alfabetos, y se nota como* ε.

El conjunto de cadenas sobre un alfabeto A excluyendo la cadena vacía se nota como A^+. Usaremos indistintamente *palabra o cadena*. Si no hay confusión, la palabra formada por un solo símbolo se representa por el propio símbolo.

La operación fundamental en el conjunto de las cadenas A^* es la concatenación.

Definición 5 *Si $u, v \in A^*$, $u = a_1 \ldots a_n$, $v = b_1 \ldots b_n$, se llama concatenación de u y v a la cadena u.v (o simplemente uv) dada por $a_1 \ldots a_n b_1 \ldots b_m$.*

La concatenación tiene las siguientes propiedades:

1. $|u.v| = |u| + |v|, \quad \forall u, v \in A^*$
2. *Asociativa.-* $u.(v.w) = (u.v).w, \quad \forall u, v, w \in A^*$
3. *Elemento Neutro.-* $u.\varepsilon = \varepsilon.u = u, \quad \forall u \in A^*$

Los propiedades asociativa y elemento neutro dotan al conjunto de las cadenas con la operación de concatenación de la estructura de monoide. La propiedad conmutativa no se verifica.

Consideramos la iteración n-sima de una cadena como la concatenación con ella misma n veces, y se define de forma recursiva:

Definición 6 *Si $u \in A^*$ entonce*

- $u^0 = \varepsilon$
- $u^{i+1} = u^i.u, \quad \forall i \geq 0$

Definición 7 *Si $u = a_1 \ldots a_n \in A^*$, entonces la cadena inversa de u es la cadena $u^{-1} = a_n \ldots a_1 \in A^*$.*

1.3.2. Lenguajes

Definición 8 *Un lenguaje sobre el alfabeto A es un subconjunto del conjunto de las cadenas sobre A: $L \subseteq A^*$.*

Los lenguajes los notaremos con las letras intermedias del alfabeto latino en mayúsculas.

Ejemplo 2 *Los siguientes son lenguajes sobre un alfabeto A, cuyo contenido asumimos conocer claramente:*

- $L_1 = \{a, b, \varepsilon\}$, *símbolos a, b y la cadena vacía*
- $L_2 = \{a^i b^i \mid i = 0, 1, 2, \ldots\}$, *palabras formadas de una sucesión de símbolos a, seguida de la misma cantidad de símbolos b.*
- $L_3 = \{uu^{-1} \mid u \in A^*\}$, *palabras formadas con símbolos del alfabeto A y que consisten de una palabra, seguida de la misma palabra escrita en orden inverso.*

- $L_4 = \{a^{n^2} \mid n = 1, 2, 3, \ldots\}$, *palabras que tienen un número de símbolos a que sea cuadrado perfecto, pero nunca nulo.*

Aparte de las operaciones de unión e intersección de lenguajes, dada su condición de conjuntos existe la operación de concatenación.

Definición 9 *If L_1, L_2 son dos lenguajes sobre el alfabeto A, la concatenación de estos dos lenguajes es el que se obtiene de acuerdo con la siguiente expresión,*

$$L_1L_2 = \{u_1u_2 \mid u_1 \in L_1, u_2 \in L_2\}$$

Ejemplo 3 *Si $L_1 = \{0^i1^i : i \geq 0\}$, $\quad L_2 = \{1^i0^i : i \geq 0\}$ entonces,*

$$L_1L_2 = \{0^i1^i1^j0^j : i, j \geq 0\}$$

Propiedades:

- $L\emptyset = \emptyset L = \emptyset$ ($\emptyset$ es el Lenguaje que contiene 0 palabras)
- *Elemento Neutro.-* $\{\varepsilon\}L = L\{\varepsilon\} = L$
- *Asociativa.-* $L_1(L_2L_3) = (L_1L_2)L_3$

La iteración de lenguajes se define como en las palabras, de forma recursiva:

Definición 10 *Si L es un lenguaje sobre el alfabeto A, entonces la iteración de este lenguaje se define de acuerdo con las siguientes expresiones recursivas,*

$$L^0 = \{\varepsilon\}$$

$$L^{i+1} = L^iL$$

Definición 11 *Si L es un lenguaje sobre el alfabeto A, la clausura de Kleene de L es el lenguaje obtenido de acuerdo con la siguiente expresión:*

$$L^* = \bigcup_{i \geq 0} L^i$$

Definición 12 *Si L es un lenguaje sobre el alfabeto A, entoces L^+ es el lenguaje dado por:*

$$L^+ = \bigcup_{i \geq 1} L^i$$

Propiedades:

- $L^+ = L^*$ si $\varepsilon \in L$
- $L^+ = L^* - \{\varepsilon\}$ si $\varepsilon \notin L$

Ejemplo 4 *Si* $L = \{0,01\}$*, entonces,* L^* *= Conjunto de palabras sobre* $\{0,1\}$ *en las que un uno va siempre precedido de un cero.*

Por otra parte, L^+ *= Conjunto de palabras sobre* $\{0,1\}$ *en las que un uno va siempre precedido de un cero y distintas de la palabra vacía.*

Definición 13 *Si L es un lenguaje, el lenguaje inverso de L es el lenguaje dado por:*

$$L^{-1} = \{u \mid u^{-1} \in L\}$$

Definición 14 *Si L es un lenguaje sobre el alfabeto A, entonces la cabecera de L es el lenguaje dado por*

$$CAB(L) = \{u \mid u \in A^* \text{ y } \exists v \in A^* \text{ tal que } uv \in L\}$$

Es decir, la CAB(L) contiene todas las palabras del lenguaje y aquellas otras que tengan como primeros caracteres, alguna palabra del mismo.

Ejemplo 5 *Si* $L = \{0^i 1^i : i \geq 0\}$*, entonces* $CAB(L) = \{0^i 1^j : i \geq j \geq 0\}$.

Definición 15 *Si* A_1 *y* A_2 *son dos alfabetos, una aplicación*

$$h : A_1^* \rightarrow A_2^*$$

se dice que es un homomorfismo si y solo si

$$h(uv) = h(u)h(v)$$

Consecuencias:

- $h(\varepsilon) = \varepsilon$
- $h(a_1 \ldots a_n) = h(a_1) \ldots h(a_n)$

Ejemplo 6 *Si* $A_1 = \{0,1,2,3,4,5,6,7,8,9\}, A_2 = \{0,1\}$
la siguiente aplicación es un homorfismo

$$h(0) = 0000, \quad h(1) = 0001, \quad h(2) = 0010, \quad h(3) = 0011$$
$$h(4) = 0100, \quad h(5) = 0101, \quad h(6) = 0110, \quad h(7) = 0111$$
$$h(8) = 1000 \quad h(9) = 1001$$

Ejemplo 7

- *Si A es un afabeto, la aplicación que transforma cada palabra $u \in A^*$ en su inversa no es un homomorfismo de A^* en A^*, ya que la transformación no se hace símbolo a símbolo.*
- *La transformación que a cada palabra sobre $\{0,1\}^*$ le añade* 00 *al principio y* 11 *al final no es un homomorfismo. Si* 0 *se transforma en 00011, entonces* 00 *se debería de transformar en* 0001100011 *si fuese un homomorfismo, y en realidad se transforma en* 000011.

1.3.3. Gramáticas Generativas

Desde un punto de vista matemático una gramática se define de la siguiente forma:

Definición 16 *Una gramática generativa es un cuadrupla (V,T,P,S) en la que*

- *V es un alfabeto, llamado de variables o símbolos no terminales. Sus elementos se suelen representar con letras mayúsculas.*
- *T es un alfabeto, llamado de símbolos terminales. Sus elementos se suelen representar con letras minúsculas.*
- *P es un conjunto de pares (α,β), llamados reglas de producción, donde $\alpha,\beta \in (V \cup T)^*$ y α contiene, al menos un símbolo de V.*

  El par (α,β) se suele representar como $\alpha \to \beta$.
- *S es un elemento de V, llamado símbolo de partida.*

La razón de notar los elementos del alfabeto V con letras mayúsculas es para no confundirlos con los símbolos terminales. Las cadenas del alfabeto $(V \cup T)$ se notan con letras griegas para no confundirlas con las cadenas del alfabeto T, que seguirán notándose como de costumbre: $u,v,x\ldots$.

Ejemplo 8 *Sea la gramática (V,T,P,S) dada por los siguientes elementos,*

- $V = \{E\}$
- $T = \{+,*,(,),a,b,c\}$

– *P está compuesto por las siguientes reglas de producción*

$$\begin{array}{lll} E \to E+E, & E \to E*E, & E \to (E), \\ E \to a, & E \to b, & E \to c \end{array}$$

– $S = E$

Una gramática se usa para generar las distintas palabras de un determinado lenguaje. Esta generación se hace mediante una aplicación sucesiva de reglas de producción comenzando por el símbolo de partida S. Las siguientes definiciones expresan esta idea de forma más rigurosa.

Definición 17 *Dada una gramática* $G = (V,T,P,S)$ *y dos palabras* $\alpha, \beta \in (V \cup T)^*$*, decimos que* β *es derivable a partir de* α *en un paso* $(\alpha \Longrightarrow \beta)$ *si y solo si existe una producción* $\gamma \to \varphi$ *tal que*

- γ *es una subcadena de* α*.*
- β *se puede obtener a partir de* α*, cambiando la subcadena* γ *for* φ*.*

Ejemplo 9 *Haciendo referencia a la gramática del ejemplo anterior, tenemos las siguientes derivaciones,*

$$E \Longrightarrow E+E \Longrightarrow (E)+E \Longrightarrow (E)+(E) \Longrightarrow (E*E)+(E) \Longrightarrow (E*E)+(E*E)$$

Definición 18 *Dada una gramática* $G = (V,T,P,S)$ *y dos palabras* $\alpha, \beta \in (V \cup T)^*$*, decimos que* β *es derivable de* α $(\alpha \overset{*}{\Longrightarrow} \beta)$*, si y solo si existe una sucesión de palabras* $\gamma_1, \ldots, \gamma_n$ $(n \geq 1)$ *tales que*
$\alpha = \gamma_1 \Longrightarrow \gamma_2 \Longrightarrow \ldots \Longrightarrow \gamma_n = \beta$

Ejemplo 10 *En el caso anterior podemos decir que* $(E*E)+(E*E)$ *es derivable a partir de* E*:*

$$E \overset{*}{\Longrightarrow} (E*E)+(E*E)$$

.

Definición 19 *Se llama lenguaje generado por una gramática* $G = (V,T,P,S)$ *al conjunto de cadenas formadas por símbolos terminales y que son derivables a partir del símbolo de partida. Es decir,*

$$L(G) = \{u \in T^* \mid S \overset{*}{\Longrightarrow} u\}$$

Ejemplo 11 *En el caso de la gramática de los ejemplos anteriores* $(E*E)+(E*E)$ *no pertenece al lenguaje generado por G, ya que hay símbolos que no son terminales. Sin embargo,* $(a+c)*(a+b)$ *si pertenece a* $L(G)$*, ya que se puede comprobar que es derivable a partir de E (símbolo de partida) y solo tiene símbolos terminales.*

Si en una gramática comenzamos a hacer derivaciones a partir del símbolo original S, dicha derivación acabará cuando solo queden símbolos terminales, en cuyo caso la palabra resultante pertenece a $L(G)$, o cuando queden variables pero no se pueda aplicar ninguna regla de producción, en cuyo caso dicha derivación no puede llevar a ninguna palabra de $L(G)$. En general, cuando estemos haciendo una derivación puede haber más de una regla de producción aplicable en cada momento.

Cuando no sea importante distinguir si la derivación de una palabra en una gramática se haya realizado en uno o varios pasos, entonces eliminaremos la $*$ del símbolo de derivación. Así, escribiremos $\alpha \Longrightarrow \beta$, en lugar de $\alpha \overset{*}{\Longrightarrow} \beta$.

Ejemplo 12 *Sea* $G=(V,T,P,S)$ *una gramática, donde* $V=\{S,A,B\}$, $T=\{a,b\}$*, las reglas de producción son*

$$\begin{array}{llll} S\rightarrow aB, & S\rightarrow bA, & A\rightarrow a, & A\rightarrow aS, \\ A\rightarrow bAA, & B\rightarrow b, & B\rightarrow bS, & B\rightarrow aBB \end{array}$$

y el símbolo de partida es S.

Esta gramática genera el lenguaje

$$L(G)=\{u \mid u\in\{a,b\}^+ \text{ y } N_a(u)=N_b(u)\}$$

donde $N_a(u)$ *y* $N_b(u)$ *son el número de apariciones de símbolos a y b, en u, respectivamente.*

Esto es fácil de ver interpretando que,

- *S genera (o produce) palabras con igual número de a que de b.*
- *A genera palabras con una a de más.*
- *B produce palabras con una b de más.*
- *S genera palabras con igual número de a que de b.*

Hay que demostrar que todas las palabras del lenguaje tienen el mismo número de a que de b, hay que probar que todas las palabras generadas cumplen esta condición y que todas las palabras que cumplen esta condición son generadas.

Para lo primero basta con considerar el siguiente razonamiento. Supongamos $N_{a+A}(\alpha)$ *y* $N_{b+B}(\alpha)$ *que son el número de a + el número de A en* α *y el número de b + el número de B en* α*, respectivamente. Entonces,*

- *Cuando se empieza a generar una palabra, comenzamos con S y tenemos la igualdad* $N_{a+A}(S) = N_{b+B}(S) = 0$.
- *También se puede comprobar que si* α' *se obtiene de* α *en un paso de derivación y* $N_{a+A}(\alpha) = N_{b+B}(\alpha)$, *entonces* $N_{a+A}(\alpha') = N_{b+B}(\alpha')$
- *Si la condición de igualdad de* N_{a+A} *y* N_{b+B} *se verifica al principio y, si se verifica antes de un paso, entonces se verifica después de aplicarlo, necesariamente se verifica al final de la derivación. Si hemos derivado u, entonces* $N_{a+A}(u) = N_{b+B}(u)$.
- *Como u no tiene variables, entonces* $N_{a+A}(u) = N_a(u)$ *y* $N_{b+B}(u) = N_b(u)$, *por lo tanto,* $N_a(u) = N_b(u)$, *es decir si u es generada contiene el mismo número de a que de b.*

Para demostrar que todas las palabras del lenguaje son generadas por la gramática, damos el siguiente algoritmo que en n pasos es capaz de generar una palabra de n símbolos. El algoritmo genera las palabras por la izquierda obteniendo, en cada paso, un nuevo símbolo de la palabra a generar.

- *Para generar una a*
  - *Si a último símbolo de la palabra, aplicar* $A \to a$
  - *Si no es el último símbolo*
    - *Si la primera variable es S aplicar* $S \to aB$
    - *Si la primera variable es B aplicar* $B \to aBB$
    - *Si la primera variable es A*
      - *Si haya más variables aplicar* $A \to a$
      - *Si no hay más, aplicar* $A \to aS$
- *Para generar una b*
  - *Si b último símbolo de la palabra, aplicar* $B \to b$
  - *Si no es el último símbolo*
    - *Si la primera variable es S aplicar* $S \to bA$
    - *Si la primera variable es A aplicar* $A \to bAA$
    - *Si la primera variable es B*
      - *Si haya más variables aplicar* $B \to b$
      - *Si no hay más, aplicar* $B \to bS$

Las condiciones que garantizan que todas las palabras son generadas mediante este algoritmo son las siguientes:

- *Las palabras generadas tienen primero símbolos terminales y después variables.*
- *Se genera un símbolo de la palabra en cada paso de derivación*
- *Las variables que aparecen en la palabra pueden ser:*
  - *Una cadena de A (si hemos generado más b que a)*
  - *Una cadena de B (si hemos generado más a que b)*
  - *Una S si hemos generado las mismas a que b*
- *Antes de generar el último símbolo tendremos como variables:*
  - *Una A si tenemos que generar a*
  - *Una B si tenemos que generar b*
- *Entonces aplicamos la primera opción para generar los símbolos y la palabra queda generada.*

Ejemplo 13 *Sea* $G = (\{S,X,Y\},\{a,b,c\},P,S)$ *donde P tiene las reglas,*

$$\begin{array}{llll} S \to abc & S \to aXbc & Xb \to bX & Xc \to Ybcc \\ bY \to Yb & aY \to aaX & aY \to aa & \end{array}$$

Esta gramática genera el lenguaje: $\{a^n b^n c^n \mid n = 1,2,\ldots\}$.

Para ver esto observemos que S en un paso, puede generar abc ó aXbc. Así que $abc \in L(G)$. *A partir de aXbc solo se puede relizar la siguiente sucesión de derivaciones,*

$$aXbc \Longrightarrow abXc \Longrightarrow abYbcc \Longrightarrow aYbbcc$$

En este momento podemos aplicar dos reglas:

– $aY \to aa$, *en cuyo caso producimos* $aabbcc = a^2b^2c^2 \in L(G)$

– $aY \to aaX$, *en cuyo caso producimos aaXbbcc*

A partir de aaXbbcc, se puede comprobar que necesariamente llegamos a $a^2Yb^3c^3$. *Aquí podemos aplicar otra vez las dos reglas de antes, produciendo* $a^3b^3c^3$ *ó* $a^3Xb^3c^3$. *Así, mediante un proceso de inducción, se puede llegar a demostrar que las únicas palabras de símbolos terminales que se pueden llegar a demostrar son* $a^nb^nc^n, n \geq 1$.

1.3.4. Jerarquía de Chomsky

De acuerdo con lo que hemos visto, toda gramática genera un único lenguaje, pero distintas gramáticas pueden generar el mismo lenguaje. Podríamos pensar en clasificar las gramáticas por el lenguaje que generan, por este motivo hacemos la siguiente definición.

Definición 20 *Dos gramáticas se dicen debilmente equivalentes si generan el mismo lenguaje.*

Sin embargo, al hacer esta clasificación nos encontramos con que el problema de saber si dos gramáticas generan el mismo lenguaje es indecidible. No existe ningún algoritmo que acepte como entrada dos gramáticas y nos diga (la salida del algoritmo) si generan o no el mismo lenguaje.

De esta forma, tenemos que pensar en clasificaciones basadas en la forma de la gramática, más que en la naturaleza del lenguaje que generan. La siguiente clasificación se conoce como jerarquía de Chomsky y sigue esta dirección.

Definición 21 *Una gramática se dice que es de*

- **Tipo 0** *Cualquier gramática. Sin restricciones.*
- **Tipo 1** *Si todas las producciones tienen la forma*

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$$

donde $\alpha_1, \alpha_2, \beta \in (V \cup T)^*, A \in V,$ *y* $\beta \neq \varepsilon$*, excepto posiblemente la regla* $S \rightarrow \varepsilon$*, en cuyo caso S no aparece a la derecha de las reglas.*

- **Tipo 2** *Si cualquier producción tiene la forma*

$$A \rightarrow \alpha$$

dode $A \in V, \alpha \in (V \cup T)^*$.

- **Tipo 3** *Si toda regla tiene la forma*

$$A \rightarrow uB \text{ ó } A \rightarrow u$$

donde $u \in T^*$ *y* $A, B \in V$

Definición 22 *Un lenguaje se dice que es de tipo* $i (i = 0, 1, 2, 3)$ *si y solo si es generado por una gramática de tipo i. La clase o familia de lenguajes de tipo i se denota por* $\mathcal{L}_i$.

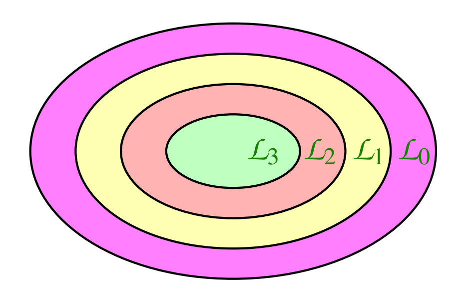


Figura 1.1: Estructura de las clases de lenguajes

Se puede demostar que $\mathcal{L}_3 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_0$.

Las gramáticas de tipo 0 se llaman también gramáticas con estructura de frase, por su origen linguístico. Los lenguajes aceptados por las gramáticas de tipo 0 son los *recursivamente enumerables*.

Las gramáticas de tipo 1 se denominan dependientes del contexto. Los lenguajes aceptados por estas gramáticas son los lenguajes *dependientes del contexto*.

Las gramáticas de tipo 2, así como los lenguajes generados, se llaman independientes del contexto.

Las gramáticas de tipo 3 se denominan regulares o de estado finito. Los lenguajes aceptados por estas gramáticas se denominan *conjuntos regulares*.

Los homorfismos son útiles para demostrar teoremas.

Teorema 1 *Para toda gramática $G = (V,T,P,S)$ podemos dar otra gramática $G' = (V',T,P',S)$ que genere el mismo lenguaje y tal que en la parte izquierda de las reglas solo aparezcan variables.*

Demostración

Si la gramática es de tipo 3 ó 2 no hay nada que demostrar.

Si la gramática es de tipo 0 ó 1, entonces para cada $a_i \in T$ introducimos una variable $A_i \notin V$. Entonces hacemos $V' = V \cup \{A_1,\ldots,A_k\}$, donde k es el número de símbolos terminales.

Ahora P' estará formado por las reglas de P donde, en todas ellas, se cambia a_i por A_i. Aparte de ello añadimos una regla $A_i \rightarrow a_i$ para cada $a_i \in$T.

Podemos ver que $L(G) \subseteq L(G')$. En efecto, si derivamos $u = a_{i_1} \ldots a_{i_n} \in L(G)$, entonces usando las reglas correspondientes, podemos derivar $A_{i_1} \ldots A_{i_n}$ en G'. Como en G' tenemos las reglas $A_i \rightarrow a_i$, entonces podemos derivar $u \in L(G')$.

Para demostrar la inclusión inversa: $L(G') \subseteq L(G)$, definimos un homomorfismo h de $(V' \cup T)^*$ en $(V \cup T)^*$ de la siguiente forma,

1. $h(A_i) = a_i, \quad i = 1, \ldots, k$
2. $h(x) = x, \quad \forall x \in V \cup T$

Ahora se puede demostrar que este homorfismo transforma las reglas: si $\alpha \rightarrow \beta \in P'$, entonces $h(\alpha) \rightarrow h(\beta)$ es una producción de P ó $h(\alpha) = h(\beta)$.

Partiendo de esto se puede demostrar que si $\alpha \Longrightarrow \beta$ entonces $h(\alpha) \Longrightarrow h(\beta)$. En particular, como consecuencia, si $S \Longrightarrow u, \quad u \in T^*$, entonces $h(S) = S \Longrightarrow h(u) = u$. Es decir, si $u \in L(G')$ entonces $u \in L(G)$. Con lo que definitivamente $L(G) = L(G')$. ■

Ejercicios

1. *Demostrar que la gramática*

$$G = (\{S\}, \{a,b\}, \{S \rightarrow \varepsilon, S \rightarrow \ aSb\}, S)$$

genera el lenguaje

$$L = \{a^i b^i \mid i = 0, 1, 2\}$$

Solución:
Si seguimos este procedimiento, nos encontramos que podemos ir generando todas las palabras de la forma $a^i b^i$, y siempre nos queda la palabra $a^i S b^i$ para seguir generando las palabras de mayor longitud.

Por otra parte, estas son las únicas palabras que se pueden generar.

2. *Encontrar el lenguaje generado por la gramática $G = (\{A,B,S\}, \{a,b\}, P, S)$ donde P contiene las siguientes producciones*

$$\begin{array}{lll} S \rightarrow aAB & bB \rightarrow a & Ab \rightarrow SBb \\ Aa \rightarrow SaB & B \rightarrow SA & B \rightarrow ab \end{array}$$

Solución:
El resultado es el Lenguaje vacío*: nunca se puede llegar a generar una palabra con símbolos terminales. Siempre que se sustituye S aparece A, y siempre que se sustituye A aparece S.*

3. *Encontrar una gramática libre del contexto para generar cada uno de los siguientes lenguajes*

   a) $L = \{a^i b^j \mid i, j \in N, i \leq j\}$
   Solución:

$$S \rightarrow aSb$$

$$S \rightarrow \varepsilon$$

$$S \rightarrow Sb$$

   b) $L = \{a^i b^j a^j b^i \mid i, j \in N\}$
   Solución:

$$S \rightarrow aSb$$

$$S \rightarrow B, \quad B \rightarrow bBa, \quad B \rightarrow \varepsilon$$

   c) $L = \{a^i b^i a^j b^j \mid i, j \in N\}$
   Solución:
   Podemos generar $\{a^i b^i \mid i \in N\}$ *con:*

$$S_1 \rightarrow aS_1b, \quad S_1 \rightarrow \varepsilon$$

   El lenguaje L se puede generar añadiendo:

$$S \rightarrow S_1S_1$$

   siendo S el símbolo inicial.

d) $L = \{a^i b^i \mid i \in N\} \cup \{b^i a^i \mid i \in N\}$
Solución:
Podemos generar $\{a^i b^i \mid i \in N\}$ *con:*

$$S_1 \rightarrow aS_1b, \quad S_1 \rightarrow \varepsilon$$

y $\{b^i a^i \mid i \in N\}$ *con*

$$S_2 \rightarrow bS_2a, \quad S_2 \rightarrow \varepsilon$$

El lenguaje L se puede generar añadiendo:

$$S \rightarrow S_1, \quad S \rightarrow S_2$$

siendo S el símbolo inicial.

e) $L = \{uu^{-1} \mid u \in \{a,b\}^*\}$
Solución:

$$S \rightarrow aSa, \quad S \rightarrow bSb, \quad S \rightarrow \varepsilon$$

f) $L = \{a^i b^j c^{i+j} \mid i, j \in N\}$
Solución:

$$S \rightarrow aSc, \quad S \rightarrow B,$$
$$B \rightarrow bBc, \quad B \rightarrow \varepsilon$$

donde $\mathcal{N}$ *es el conjunto de los números naturales incluyendo el 0.*

4. *Determinar si la gramática* $G = (\{S,A,B\},\{a,b,c,d\},P,S)$ *donde* P *es el conjunto de reglas de producción*

$$\begin{array}{lll} S \rightarrow AB & A \rightarrow Ab & A \rightarrow a \\ B \rightarrow cB & B \rightarrow d & \end{array}$$

genera un lenguaje de tipo 3.

Solución:

Esta gramática genera el lenguaje: $\{ab^ic^jd : i,j \in N\}$*, y este lenguaje se puede generar mediante la gramática:*

$S \rightarrow aB, \quad B \rightarrow bB, \quad B \rightarrow C, \quad C \rightarrow cC, \quad C \rightarrow d$

Como esta gramática es de tipo 3, el lenguaje lo es.

Capítulo 2

Autómatas Finitos, Expresiones Regulares y Gramáticas de tipo 3

Los autómatas finitos son capaces de reconocer solamente, un determinado tipo de lenguajes, llamados Lenguajes Regulares, que pueden ser caracterizados también, mediante un tipo de gramáticas llamadas también regulares. Una forma adicional de caracterizar los lenguajes regulares, es mediante las llamadas expresiones regulares, que son las frases del lenguaje, construidas mediante operadores sobre el alfabeto del mismo y otras expresiones regulares, incluyendo el lenguaje vacío.

Estas caracterizaciones de los lenguajes regulares se utilizan en la práctica, según que la situación concreta esté favorecida por la forma de describir el lenguaje de cada una de ellas. Los autómatas finitos se utilizan generalmente para verificar que las cadenas pertenecen al lenguaje, y como un analizador en la traducción de algorítmos al ordenador.

Las gramáticas y sus reglas de producción se usan frecuentemente en la descripción de la sintaxis de los lenguajes de programación que se suele incluir en los manuales correspondientes. Por otro lado, las expresiones regulares proporcionan una forma concisa y relativamente sencilla (aunque menos intuitiva) para describir los lenguajes regulares, poniendo de manifiesto algunos detalles de su estructura que no quedan tan claros en las otras caracterizaciones. Su uso es habitual en editores de texto, para búsqueda y sustitución de cadenas.

En definitiva, las caracterizaciones señaladas de los lenguajes (formales) regulares, y por tanto ellos mismos, tienen un uso habitual en la computación práctica actual. Esto por sí mismo justificaría su inclusión en un curriculum de computación teórica.

2.1. Autómatas Finitos Determinísticos

Antes de dar una definición formal de lo que es un autómata vamos a dar una descripción intuitiva mediante un ejemplo.

Ejemplo 14 *Vamos a diseñar un autómata que reconozca el paso de un alumno por una asignatura, por ejemplo, Modelos de Computación I. Representará las distintas decisiones que se realizan y si se aprueba o suspende la asignatura. Se controla que no haya más de dos convocatorias por año y se termina cuando se aprueba la asignatura.*

Habrá un alfabeto de entrada contendrá los siguientes elementos:

- P: *El alumno se presenta a un examen.*
- N: *El alumno no se presenta a un examen.*
- A: *El alumno aprueba un examen.*
- S: *El alumno suspende un examen.*

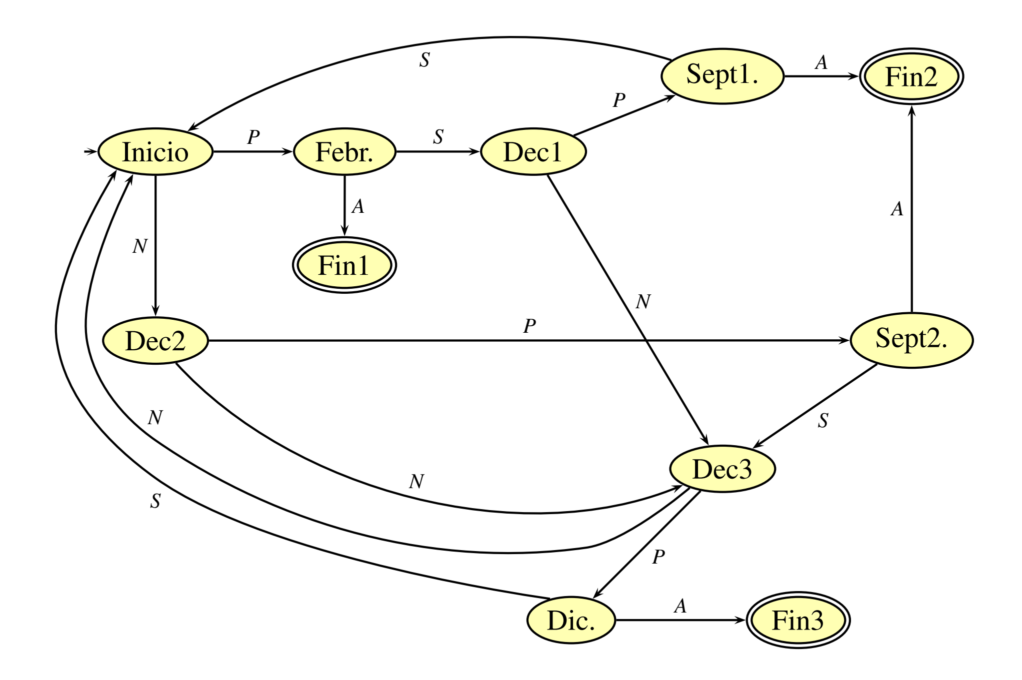


Figura 2.1: Paso de un alumno por una asignatura

La secuencia se ilustra en la figura 2.1. Comenzamos en un estado, Inicio. A continuación decidimos si presentarnos en Febrero o no. Si nos presentamos y aprobamos, terminamos. Si no nos presentamos o suspendemos, tenemos que decidir si nos presentamos en septiembre, pero como hay que controlar que un estudiante no se presente a tres convocatorias en un año, los estados son distintos en ambos casos. Si en septiembre aprobamos, terminamos. Si suspendemos y ya nos habíamos presentado en febrero, comenzamos de nuevo. En otro caso, podemos decidir si presentarnos en diciembre. Si aprobamos, terminamos y si suspendemos, empezamos de nuevo.

Este esquema corresponde a un autómada finito. Se caracteriza por una estructura de control que depende de un conjunto finito de estados. Se pasa de unos a otros leyendo símbolos del alfabeto de entrada. Este autómata representa una versión simplificada del problema real, ya que no controla el número total de convocatorias. Un autómata para todas las asignaturas se podría construir uniendo autómatas para cada una de las asignaturas, pero teniendo en cuenta relaciones como requisitos entre las mismas.

Definición 23 *Un autómata finito es una quintupla* $M = (Q, A, \delta, q_0, F)$ *en que*

– Q *es un conjunto finito llamado conjunto de estados*

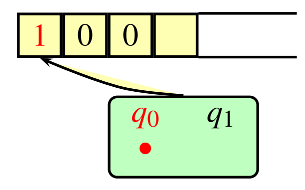


Figura 2.2: Autómata de Estado Finito

– *A es un alfabeto llamado alfabeto de entrada*

– *δ es una aplicación llamada función de transición*

$$\delta : Q \times A \rightarrow Q$$

– *q_0 es un elemento de Q, llamado estado inicial*

– *F es un subconjunto de Q, llamado conjunto de estado finales.*

Desde el punto de vista intuitivo, podemos ver un autómata finito como una caja negra de control (ver Figura 2.2), que va leyendo símbolos de una cadena escrita en una cinta, que se puede considerar ilimitada por la derecha. Existe una cabeza de lectura que en cada momento esá situada en una casilla de la cinta. Inicialmente, esta se sitúa en la casilla de más a la izquierda. El autómata en cada momento está en uno de los estado de Q. Inicialmente se encuentra en q_0.

En cada paso, el autómata lee un símbolo y según el estado en que se encuentre, cambia de estado y pasa a leer el siguiente símbolo. Así sucesivamente hasta que termine de leer todos los símbolos de la cadena. Si en ese momento la máquina está en un estado final, se dice que el autómata acepta la cadena. Si no está en un estado final, la rechaza.

Definición 24 *El diagrama de transición de un Autómata de Estado Finito es un grafo en el que los vértices representan los distintos estados y los arcos las transiciones entre los estados. Cada arco va etiquetado con el símbolo que corresponde a dicha transición. El estado inicial y los finales vienen señalados de forma especial (por ejemplo, con un ángulo el estado inicial y con un doble círculo los finales).*

Ejemplo 15 *Supongamos el autómata $M = (Q, A, q_0, \delta, F)$ donde*

– $Q = \{q_0, q_1, q_2\}$

– $A = \{a, b\}$

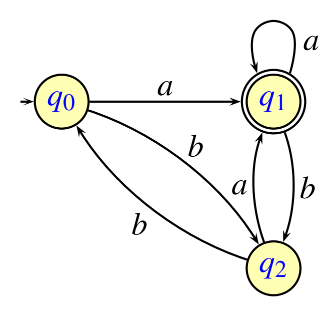


Figura 2.3: Diagrama de Transición Asociado a un Autómata de Estado Finito

– *La función de transición* δ *está definida por las siguientes igualdades:*

$$\begin{array}{ll} \delta(q_0,a) = q_1 & \delta(q_0,b) = q_2 \\ \delta(q_1,a) = q_1 & \delta(q_1,b) = q_2 \\ \delta(q_2,a) = q_1 & \delta(q_2,b) = q_0 \end{array}$$

– $F = \{q_1\}$

El diagrama de transición viene expresado en la Figura 2.3.

2.1.1. Proceso de cá lculo asociado a un Autómata de Estado Finito

Para describir formalmente el comportamiento de un autómata de estado finito, vamos a introducir el concepto de configuración y paso de cálculo.

Definición 25 *Si* $M = (Q,A,\delta,q_0,F)$ *es un autómata de estado finito una configuración es un elemento del producto cartesiano* $Q \times A^*$.

Una configuración es un par (q,u) donde q es un estado y u una palabra. Intuitivamente una configuración contiene los elementos que determinan la evolución futura del autómata. En este sentido, q será el estado en el que se encuentra el autómata y u lo que queda por leer en un momento dado.

Inicialmente, autómata está en el estado inicial q_0 y nos queda por leer toda la palabra de entrada. El siguiente concepto recoge esta idea.

Definición 26 *Si* $M = (Q,A,\delta,q_0,F)$ *es una autómata finito determinista y* $u \in A^*$ *es una palabra, entonces se llama configuración inicial asociada a esa palabra a la configuración* (q_0,u).

Intuitivamente, la configuración inicial indica que estamos en el estado inicial y nos queda por leer toda la palabra u.

A continuación definimos la relación paso de cálculo entre dos configuraciones.

Definición 27 *Si $M = (Q, A, \delta, q_0, F)$ es una autómata finito determinista y $(p,u),(q,v)$ son dos configuraciones, decimos que se puede pasar de (p,u) a (q,v) en un paso de cálculo, lo que se nota como $(p,u) \vdash (q,v)$ si y sólo si $u = av$, donde $a \in A$ y $\delta(p,a) = q$.*

Finalmente, definimos la relación de cálculo entre dos configuraciones.

Definición 28 *Si $M = (Q, A, \delta, q_0, F)$ es una autómata finito determinista y $(p,u),(q,v)$ son dos configuraciones, decimos que puede pasar de (p,u) a (q,v) en una secuencia de cálculo, lo que se nota como $(p,u) \overset{*}{\vdash} (q,v)$ si y solo si existe una sucesión de configuraciones: $(p_0,u_0),\ldots(p_n,u_n)$ donde $n \geq 1$, de tal forma que $(p_0,u_0) = (p,u)$, $\quad (p_n,u_n) = (q,v)$ y $(p_i,u_i) \vdash (p_{i+1},u_{i+1}), \forall i < n$.*

Existe otro enfoque alternativo para definir formalmente el cálculo asociado a un autómata de estado finito y consiste en definir la función de transición de un autómata aplicada a una palabra. Esta función que llamaremos δ' se define de forma recursiva

Definición 29 *Si $M = (Q, A, \delta, q_0, F)$ es un autómata de estado finito se define la función de transición asociada a palabras, como la función*

$$\delta' : Q \times A^* \to Q$$

dada por

$$\delta'(q,\varepsilon) = q$$
$$\delta'(q,aw) = \delta'(\delta(q,a),w), \quad w \in A^*,\ a \in A$$

Ejemplo 16 *En el caso del autómata del ejemplo 15, tenemos que*

$$\delta'(q_0,aba) = \delta'(\delta(q_0,a),ba) = \delta'(q_1,ba) = \delta'(\delta(q_1,b),a) =$$
$$\delta'(q_2,b) = \delta'(\delta(q_2,b),\varepsilon) = \delta'(q_1,\varepsilon) = q_1$$

Desde el punto de vista intuitivo, δ representa el comportamiento del autómata en un paso de cálculo, ante la lectura de un carácter. Esto viene dado por el estado al que evoluciona el autómata. La cabeza de lectura siempre se mueve a la derecha. δ' representa una sucesión de cálculos del autómata: a qué estado evoluciona después de haber leído una cadena de caracteres.

La relación entre δ' y $\overset{*}{\vdash}$ es la siguiente: $\delta'(p,u) = q$ si y sólo si $(p,u) \overset{*}{\vdash} (q,\varepsilon)$.

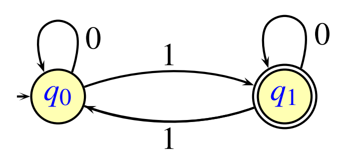


Figura 2.4: Autómata que acepta el lenguaje de palabras con un número impar de unos

2.1.2. Lenguaje aceptado por un Autómata de Estado Finito

Definición 30 *Una palabra* $u \in A^*$ *se dice aceptada por un autómata* $M = (Q,A,\delta,q_0,F)$ *si y solo si existe un* $q \in F$ *tal que* $(q_0,u) \overset{*}{\vdash} (q,\varepsilon)$. *En caso contrario, se dice que la palabra es rechazada por el autómata.*

Es decir, una palabra es aceptada por un autómata si comenzando a calcular en el estado q_0 y leyendo los distintos símbolos de la palabra llega a un estado final.

En el caso del autómata del ejemplo 15, la palabra *aba* es aceptada por el autómata. La palabra *aab* es rechazada.

Definición 31 *Dado un autómata* $M = (Q,A,\delta,q_0,F)$ *se llama lenguaje aceptado o reconocido por dicho autómata al conjunto de las palabras de* A^* *que acepta:*

$$L(M) = \{u \in A^* : (q_0,u) \overset{*}{\vdash} (q,\varepsilon), q \in F\}$$

Ejemplo 17 *El autómata dado por el diagrama de transición de la Figura 2.4 acepta el lenguaje formado por aquellas cadenas con un número impar de unos.*

Ejemplo 18 *Los autómatas suelen servir para reconocer constantes o identificadores de un lenguaje de programación. Supongamos, por ejemplo, que los reales vienen definidos por la gramática* $G = (V,T,P,S)$ *en la que*

- $T = \{+,-,E,0,1,\ldots,9,.\}$
- $V = \{<Signo>,<Digito>,<Natural>,<Entero>,<Real>\}$
- $S = <Real>$
- *P contiene las siguientes producciones*

$$<Signo> \rightarrow +|-$$
$$<Digito> \rightarrow 0|1|2|3|4|5|6|7|8|9$$

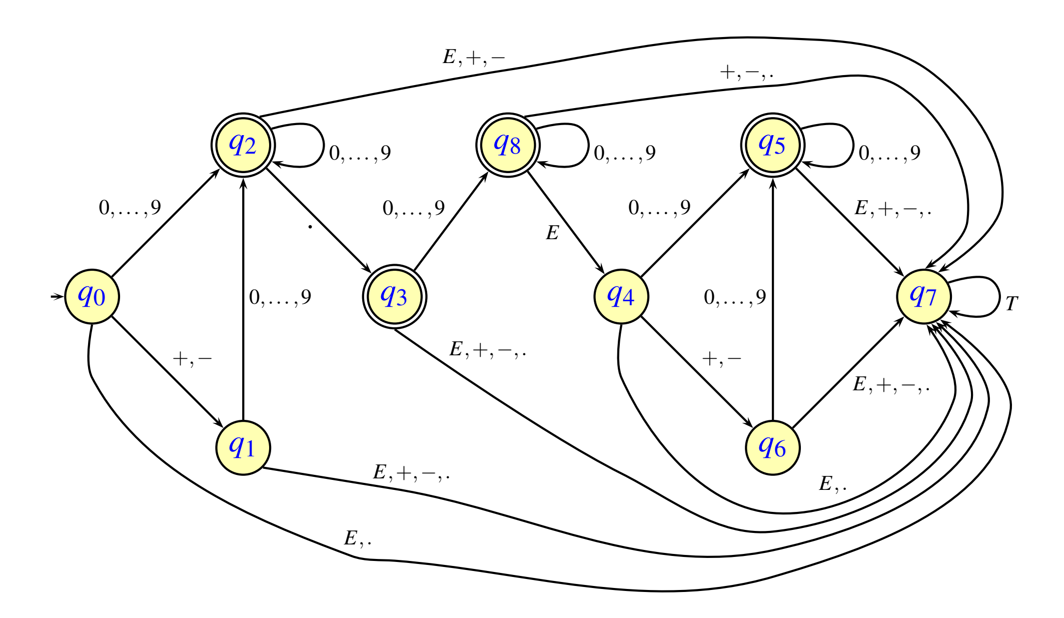


Figura 2.5: Autómata Finito Determinista que acepta constantes reales

<Natural>→<Digito> | <Digito><Natural>

<Entero>→<Natural>

<Entero>→<Signo><Natural>

<Real>→<Entero>

<Real>→<Entero>.

<Real>→<Entero> . <Natural>

<Real>→<Entero> . <Natural> E <Entero>

Este lenguaje puede ser descrito también por un autómata de estado finito, como el de la Figura 18, en el que el alfabeto es $A = \{0,1,2,\ldots,9,+,-,E\}$.

Si un lenguaje es aceptado por un autómata de estado finito, tenemos un procedimiento algorítmico sencillo que nos permite decir qué palabras pertenecen al lenguaje y qué palabras no pertenecen. Es mucho más cómodo que la gramática inicial, aunque la gramática es más expresiva. No todos los lenguajes pueden describirse mediante autómatas de estado finito. Los más importantes son los identificadores y constantes de los lenguajes. Sin embargo, es importante señalar que a partir de una gramática de tipo 3 podemos construir de forma algorítmica un autómata que acepta el mismo lenguaje que el generado por la gramática.

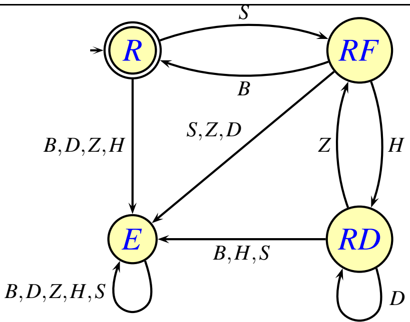


Figura 2.6: Recepción de Ficheros en el protocolo Kermit

Ejemplo 19 Comunicaciones. Protocolo Kermit.- *Los autómatas finitos proporcionan modelos de programación para numerosas aplicaciones. El protocolo Kermit se puede implementar como un control de estado finito. Vamos a fijarnos en una pequeña parte del mismo: la transferencia de ficheros entre* micros *y* mainframes *y, más concretamente en la recepción de estos ficheros.*

El esquema del programa de recepción de datos corresponde al de una máquina de estado finito (ver Figura 2.6). Inicialmente, la máquina está en un estado de espera, R. En este estado está preparado para recibir una cabecera de transmisión (símbolo de entrada S), en cuyo caso pasa al estado RF, en el que espera la recepción de una cabecera de fichero (símbolo de entrada H). En ese momento pasa al estado RD, en el que procesa una serie de datos correspodientes al fichero (símbolos D). Si, en un momento dado, recibe un fin de fichero (símbolo Z) pasa al estado RF, donde puede recibir otra cabecera de fichero. Si estando en el estado RF se recibe un código de fin de la transmisión el autómata pasa al estado inicial R. El estado final es R: Una transmisión es aceptada si, al final de cada fichero se produce un fin de fichero y al final de todos los ficheros un fin de transmisión que le permite volver a R. Si, en un momento dado se produce una entrada inesperada (por ejemplo, un fin de la transmisión cuando se están recibiendo datos de un fichero en particular) se pasa a un estado de error, E. Este estado es absorvente: cualquier otra entrada le hace quedar en el mismo estado.

El esquema del emisor es mas complicado. También ésto es solo un esquema del comportamiento del receptor. En realidad, éste no solo determina si la transmisión se ha realizado con éxito, sino que ejecuta acciones cada vez que recibe datos. Esto no lo puede hacer un autómata de estado finito, pero sí otro tipo de máquinas un poco más sofisticadas que estudiaremos más adelante. De todas formas, lo importante es advertir como el modelo de computación de los autómatas finitos sirve para expresar de forma elegante el mecanismo de control de muchos programas.

2.2. Autómatas Finitos No-Determinísticos (AFND)

En este apartado vamos a introducir el concepto de autómatas finitos no determinísticos (AFND) y veremos que aceptan exactamente los mismos lenguajes que los autómatas determinísticos. Sin embargo, serán importantes para demostrar teoremas y por su más alto poder expresivo.

En la definición de autómata no-determinístico lo único que cambia, respecto a la definición de autómata determinístico es la función de transición. Antes estaba definida

$$\delta : Q \times A \rightarrow Q$$

y ahora será una aplicación de $Q \times A$ en $\wp(Q)$ (subconjuntos de Q):

$$\delta : Q \times A \rightarrow \wp(Q)$$

La definición completa quedaría comos sigue.

Definición 32 *Un autómata finito no-determinístico es una quintupla* $M = (Q, A, \delta, q_0, F)$ *en que*

- Q *es un conjunto finito llamado conjunto de estados*
- A *es un alfabeto llamado alfabeto de entrada*
- δ *es una aplicación llamada función de transición*

$$\delta : Q \times A \rightarrow \wp(Q)$$

- q_0 *es un elemento de* Q*, llamado estado inicial*
- F *es un subconjunto de* Q*, llamado conjunto de estado finales.*

La interpretación intuitiva es que ahora el autómata, ante una entrada y un estado dado, puede evolucionar a varios estados posibles (incluyendo un solo estado o ninguno si $\delta(q, a) = \emptyset$). Es decir es como un algoritmo que en un momento dado nos deja varias opciones posibles o incluso puede no dejarnos ninguna.

Una palabra se dice aceptada por un AFND si, siguiendo en cada momento alguna de las opciones posibles, llegamos a un estado final.

A continuación, vamos a extender δ definida en $Q \times A$ a una función δ^* definida en $\wp(Q) \times A$, es decir, para que se pueda aplicar a conjuntos de estados, no sólo a un estado dado. δ^* se define de la siguiente forma,

$$\text{Si } P \subseteq Q \text{ y } u \in A, \delta^*(P,a) = \bigcup_{q \in P} \delta(q,a)$$

Esta función se puede extender para aplicarse también a palabras de la siguiente forma:
Si $B \subseteq Q$,

- $\delta^*(B,\varepsilon) = B$
- $\delta^*(B,au) = \delta^*(\delta^*(B,a),u)$

δ^* se puede aplicar también a un sólo estado de acuerdo con la siguiente expresión: $\delta^*(q,u) = \delta^*(\{q\},u)$

Con estos elementos podemos definir cuando una palabra es aceptada por un autómata no-determinístico.

Definición 33 *Sea $M = (Q,A,\delta,q_0,F)$ un autómata finito no-determinístico y $u \in A^*$. Se dice que la palabra u es aceptada por M si y solo si $\delta'(q_0,u) \cap F \neq \emptyset$.*

Definición 34 *Sea $M = (Q,A,\delta,q_0,F)$ un AFND, se llama lenguaje aceptado por el autómata al conjunto de palabras de A^* que acepta, es decir*

$$L(M) = \{u \in A^* \mid \delta'(q_0,u) \cap F \neq \emptyset\}$$

Cuando no haya lugar a confusión, δ^* será representadas como δ, simplemente.

En clase, el lenguaje aceptado por un autómata no determinista se ha definido de forma distinta, pero ambas definiciones son totalmente equivalentes. Se define en función de la definición de la relación de proceso de cálculo entre dos configuraciones que, análogamente como se hizo con los autómatas deterministas, se basa en definir los siguientes elementos:

- **Descripción Instantánea o Configuración:**

  Un elemento de $Q \times A^*$: (q,u).

- **Configuración Inicial para** $u \in A^*$**:** (q_0,u)

- **Relación paso de cálculo entre dos configuraciones:**

  $$((q,au) \vdash (p,v)) \Leftrightarrow p \in \delta(q,a))$$

  Aquí, al contrario de lo que ocurría en los autómatas deterministas, desdee una configuración se puede pasar a varias configuraciones distintas en un paso de cálculo, e incluso a ninguna.

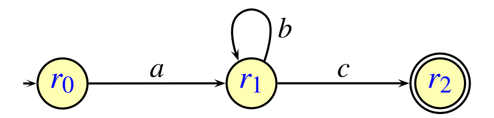


Figura 2.7: Autómata Finito No-Determinístico

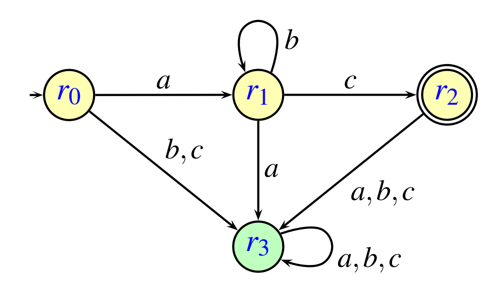


Figura 2.8: Autómata Finito Determinista

- **Relación de cálculo entre dos configuraciones:**

  $((q,u) \overset{*}{\vdash} (p,v))$ si y solo si existe una sucesión de configuraciones $C_0,\ldots,C_n$ tales que $C_0 = (q,u), C_n = (p,v)$ y $\forall i \leq n-1, C_i \vdash C_{i+1}$.

Finalmente, el lenguaje aceptado por un AF no-determinista es

$$L(M) = \{u \in A^* : \exists q \in F, (q_0,u) \overset{*}{\vdash} (q,\varepsilon)\}$$

2.2.1. Diagramas de Transición

Los diagramas de transición de los AFND son totalmente análogos a los de los autómatas determinísticos. Solo que ahora no tiene que salir de cada vértice un y solo un arco para cada símbolo del alfabeto de entrada. En un autómata no determinístico, de un vértice pueden salir ninguna, una o varias flechas con la misma etiqueta.

Ejemplo 20 *Un AFND que acepta el lenguaje*

$$L = \{u \in \{a,b,c\}^* \mid u = ab^i c, i \geq 0\}$$

es el de la Figura 2.7.

Este es un autómata no-determinista ya que hay transiciones no definidas.

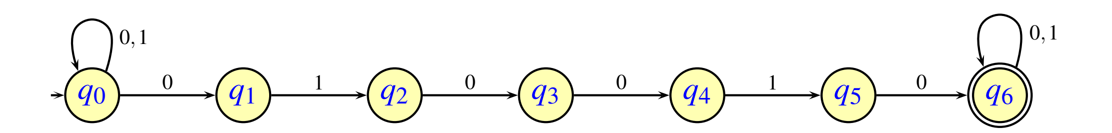


Figura 2.9: Autómata No-Determinista que reconoce la cadena 010010.

En general, los autómatas no-determinísticos son más simples que los determinísticos. Por ejemplo, un autómata determinístico que acepta el mismo lenguaje que el anterior es el que viene dado por la Figura 2.8

Ejemplo 21 (Reconocimiento de Patrones).- *Supongamos un ejemplo de tranmisión de datos entre barcos. El receptor de un barco debe de estar siempre esperando la trasmisión de datos que puede llegar en cualquier momento. Cuando no hay trasmisión de datos hay un ruído de fondo (sucesión aleatoria de* $0,1$*). Para comenzar la trasmisión se manda una cadena de aviso, p.e.* 010010*. Si esa cadena se reconoce hay que registrar los datos que siguen.*

El programa que reconoce esta cadena puede estar basado en un autómata finito. La idea es que este no pueda llegar a un estado no final mientras no se reciba la cadena inicial. En ese momento el autómata pasa a un estado final. A partir de ahí todo lo que llegue se registra. Nuestro propósito es hacer un autómata que llegue a un estado final tan pronto como se reconozca 010010*. Intentar hacer un autómata finito determinístico directamente puede ser complicado, pero es muy fácil el hacer un AFND, como el de la Figura 2.9.*

Hay que señalar que esto sería solamente el esquema de una sola parte de la trasmisión. Se podría complicar incluyendo también una cadena para el fin de la trasmisión.

Ejemplo 22 Autómata no-determinístico que acepta constantes reales *Ver Figura 2.10*

2.2.2. Equivalencia de Autómatas Determinísticos y No-Determinísticos

La siguiente definición muestra como construir a partir de un autómata finito no-determinístico otro determinístico que acepte el mismo lenguaje.

Definición 35 *Dado un AFND* $M = (Q,A,\delta,q_0,F)$ *se llama autómata determinístico asociado a M, al autómata* $\bar{M} = (\bar{Q},A,\bar{\delta},\bar{q}_0,\bar{F})$ *dado por*

- $\bar{Q} = \mathcal{P}(Q)$
- $\bar{q}_0 = \{q_0\}$
- $\bar{\delta}(A,a) = \delta^*(A,a)$

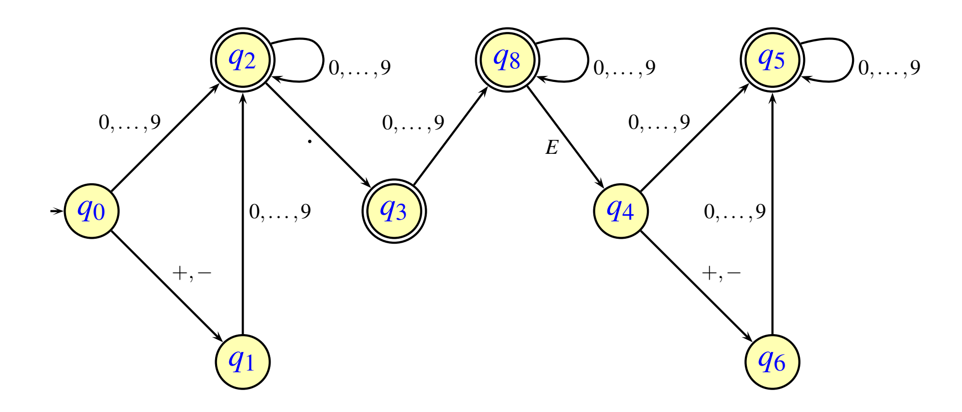


Figura 2.10: Autómata No-Determinístico que reconoce constantes reales.

– $\bar{F} = \{A \in \mathcal{P}(Q) \mid A \cap F \neq \emptyset\}$

Dado un autómata no determinístico se le hace corresponder uno determinístico que recorre todos los caminos al mismo tiempo.

Ejemplo 23 *En la Figura 2.11 podemos ver un autómata finito no-determinista del ejemplo 21 (figura 2.9) y su autómata determinístico asociado.*

Teorema 2 *Un AFND M y su correspondiente Autómata determinístico $\bar{M}$ aceptan el mismo lenguaje.*

Demostración

La demostración se basa en probar que

$$\bar{\delta}^*(\bar{q}_0, u) = \delta^*(q_0, u)$$

■

Es decir, que el conjunto de estados en los que puede estar el autómata no determinista coincide con el estado en el que está el autómata determinista que hemos construido. Ambos aceptan una palabra si al final de su lectura uno de estos estados es final.

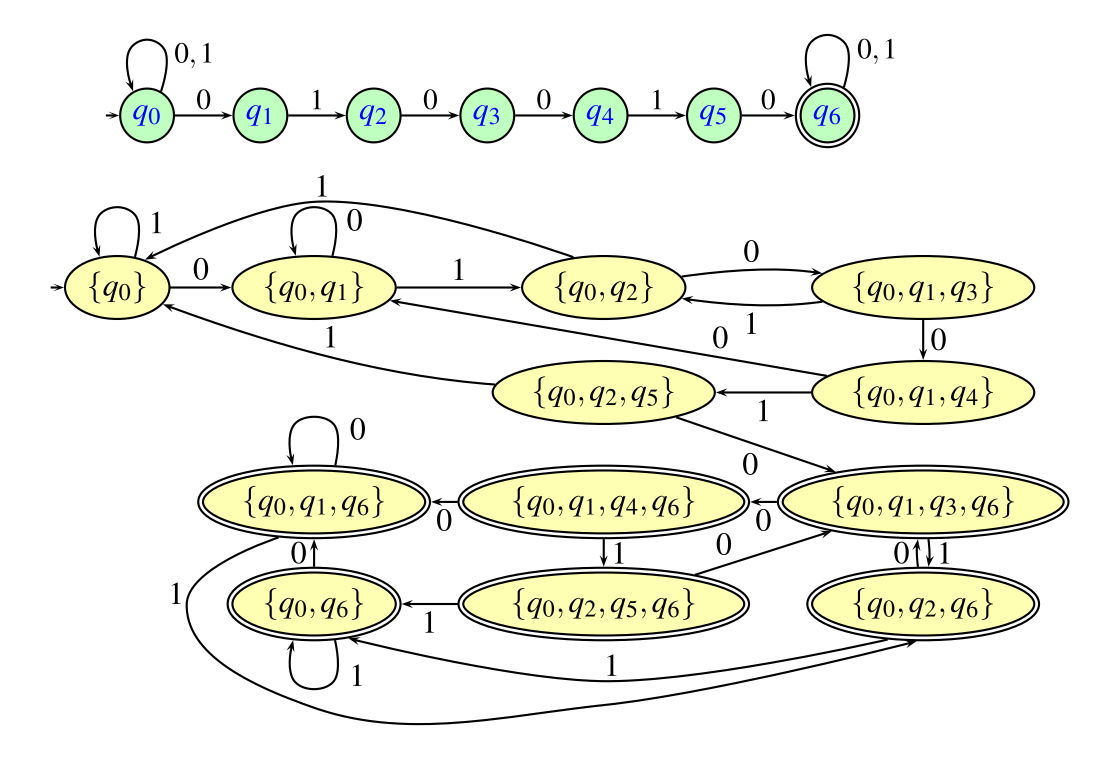


Figura 2.11: Automata Finito No-Determinístico y Autómata Finito Determinístico asociado

El teorema inverso es evidente: dado un autómata determinístico $M = (Q, A, \delta, q_0, F)$ podemos construir uno no-determinista $\bar{M} = (Q, A, \bar{\delta}, q_0, F)$ que acepta el mismo lenguaje. Solo hay que considerar que $\bar{\delta}(q_0, a) = \{\delta(q_0, a)\}$: En cada situación solo hay un camino posible, el dado por δ.

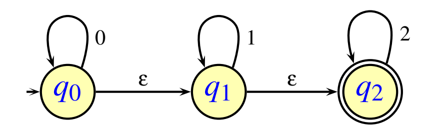
2.3. Autómatas Finitos No Deterministicos con transiciones nulas

Son autómatas que pueden relizar una transición sin consumir entrada. Estas transiciones se etiquetan con ε en el diagrama asociado.

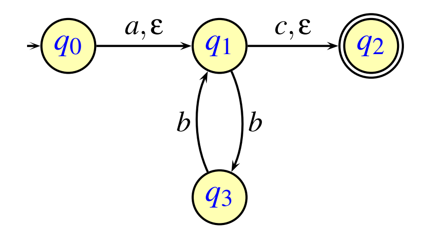
En la definición, lo único que hay que cambiar es la función de transición que ahora está definida de $Q \times (A \cup \{\varepsilon\})$ en $\mathcal{P}(Q)$.

Las transiciones nulas dan una nueva capacidad al autómata. Si se tiene una transición nula, el autómata puede quedarse donde está o cambiar de estado sin consumir ningún símbolo de la palabra de entrada. Como en los AFND, una palabra será aceptada si se llega a un estado final con alguna de las elecciones posibles.

Ejemplo 24 *El autómata siguiente acepta las palabras que son una sucesión de ceros, seguida de una sucesión de 1, seguida de una sucesión de 2.*



Ejemplo 25 *El autómata dado por el siguiente diagrama,*



acepta el lenguaje $\{b^2i : i \geq 0\} \cup \{ab^2i : i \geq 0\} \cup \{b^2ic : i \geq 0\} \cup \{ab^2ic : i \geq 0\}$

Definición 36 *Se llama clausura de un estado al conjunto de estados a los que puede evolucionar sin consumir ninguna entrada,*

$$CL(p) = \{q \in Q : \exists q_1, \ldots, q_k \text{ tal que } p = q_1, q = q_k, q_i \in \delta(q_{i-1}, \varepsilon), i \geq 2\}$$

Definición 37 *Si $P \subseteq Q$ se llama clausura de P a*

$$CL(P) = \bigcup_{p \in P} CL(p)$$

Ejemplo 26 *En el ejemplo anterior tenemos que*

$$CL(q_1) = \{q_1, q_2\}$$

$$CL(q_0, q_1) = \{q_0, q_1, q_2\}$$

La función de transición δ se extiende a conjuntos de estados de la siguiente forma:

$$\delta(R, u) = \bigcup_{q \in R} \delta(q, u)$$

A continuación vamos a determinar la función δ' que asocia a un estado y a una palabra de entrada el conjunto de los estados posibles.

$$\delta'(q, \varepsilon) = CL(q)$$

$$\delta'(q, au) = \bigcup_{r \in \delta(CL(q), a)} \delta'(r, u)$$

Es conveniente extender la función δ' a conjuntos de estados,

$$\delta'(R, u) = \bigcup_{q \in R} \delta'(q, u)$$

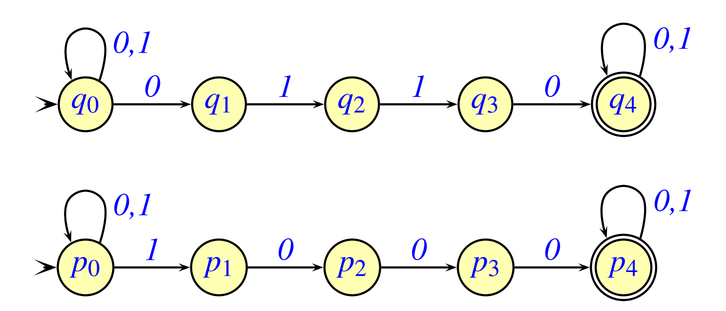
Notemos que $\delta'(q, a)$ no es lo mismo que $\delta(q, a)$ por lo que distinguiremos entre las dos funciones.

Una palabra, $u \in A^*$, se dice aceptada por un AFND con transiciones nulas $M = (Q, A, \delta, q_0, F)$ si y solo si

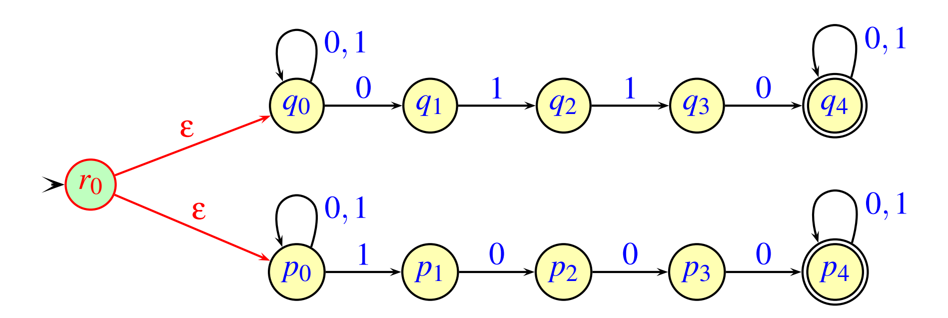
$$\delta'(q, u) \cap F \neq \emptyset$$

El lenguaje aceptado por el autómata, $L(M)$, es el conjunto de palabras de A^* que acepta. Las transiciones nulas son muy útiles para modificar autómatas o para construir autómatas mas complejos partiendo de varios autómatas.

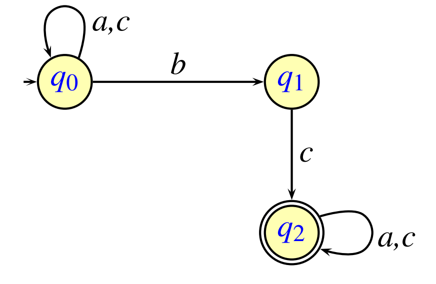
Ejemplo 27 *Los siguientes dos autómatas aceptan las palabras que contienen en su interior la subcadena* 0110 *y* 1000 *respectivamente*



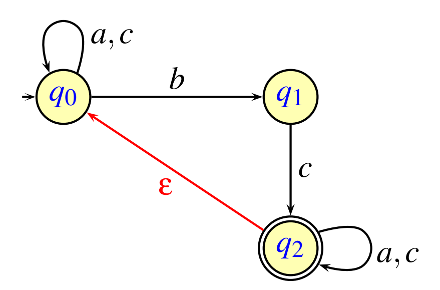
Para construir un autómata que acepte las palabras aceptadas por uno cualquiera de estos dos autómatas, es decir, las palabras con una subcadena 0110 *ó una subcadena* 1000 *se pone un nuevo estado inicial que se une con los antiguos mediante transiciones nulas.*



Ejemplo 28 *Consideremos el siguiente autómata que acepta las palabras del alfabeto* $\{a, b, c\}$ *que tienen una sola b y ésta va seguida de una c.*



Si L es el lenguaje que acepta el autómata anterior modifiquémoslo para que acepte cualquier sucesión no nula de palabras de L: cualquier palabra de L^+. *Para ello se unen los estados finales del autómata con el estado inicial mediante una transición nula:*



Las transiciones nulas, como vemos dan más capacidad de expresividad a los autómatas, pero el conjunto de lenguajes aceptados es el mismo que para los autómatas finitos determinísticos, como queda expresado en el siguiente teorema.

Teorema 3 *Para todo autómata AFND con transiciones nulas existe un autómata finito determinístico que acepta el mismo lenguaje.*

Demostración.- Si $M = (Q, A, \delta, q_0, F)$ es un AFND con transiciones nulas, basta construir un AFND sin transiciones nulas $M = (Q, A, \overline{\delta}, \overline{q}_0, \overline{F})$. A partir de éste se puede construir uno determinístico.

El autómata $\overline{M}$ se define de la siguiente forma

$$\overline{F} = \begin{cases} F & \text{if } CL(q_0) \cap F = \emptyset \\ F \cup \{q_0\} & \text{en caso contrario} \end{cases} \tag{2.1}$$

$$\overline{\delta}(q, a) = \delta'(q, a) \tag{2.2}$$

El resto de los elementos se definen igual que en M.

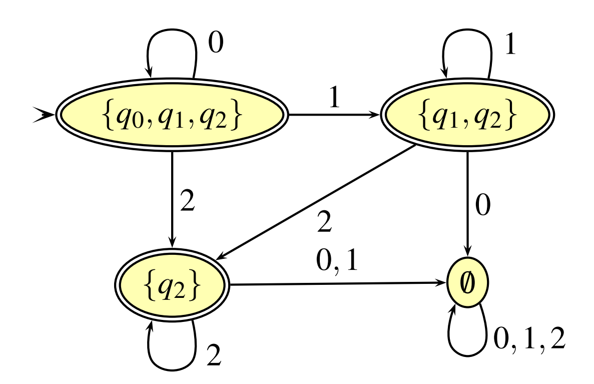
Para demostrar que ambos autómatas aceptan el mismo lenguaje basta con probar que

$$\overline{\delta}'(q, u) = \delta'(q, u) \text{ si } u \neq \varepsilon \tag{2.3}$$

y que ε se acepta de igual forma en las dos máquinas.

■

Ejemplo 29 *El AFND correspondiente al autómata con transiciones nulas del ejemplo 19, acepta el mismo lenguaje que el siguiente autómata determinista:*



2.4. Expresiones Regulares

Una expresión regular es una forma de representar cierto tipo de lenguajes sobre un determinado alfabeto. Veremos que son exactamente los aceptados por los autómatas de estado finito.

Definición 38 *Si A es un alfabeto, una expresión regular sobre este alfabeto se define de la siguiente forma:*

- $\emptyset$ *es una expresión regular que denota el lenguaje vacío.*
- ε *es una expresión regular que denota el lenguaje* $\{\varepsilon\}$
- *Si* $a \in A$, $\mathbf{a}$ *es una expresión regular que denota el lenguaje* $\{a\}$
- *Si* $\mathbf{r}$ *y* $\mathbf{s}$ *son expresiones regulares denotando los lenguajes R y S entonces definimos las siguientes operaciones:*
    - *Unión:* $(\mathbf{r}+\mathbf{s})$ *es una expresión regular que denota el lenguaje* $R \cup S$
    - *Concatenación:* $(\mathbf{rs})$ *es una expresión regular que denota el lenguaje RS*
    - *Clausura:* $\mathbf{r}^*$ *es una expresión regular que denota el lenguaje* R^*.

De acuerdo con la definición anterior, se puede determinar no solo cuales son las expresiones regulares sobre un determinado alfabeto, sino también cuales son los lenguajes que denotan o lenguajes asociados. Los lenguajes que pueden representarse mediante una expresión regular se llaman lenguajes regulares. Estos coinciden con los aceptados por los autómatas finitos, como veremos mas adelante.

Los paréntesis se pueden eliminar siempre que no haya dudas. La precedencia de las operaciones es

- Clausura

– Concatenación

– Unión

Ejemplo 30 *Si* $A = \{a, b, c\}$

– $(\mathbf{a} + \varepsilon)\mathbf{b}^*$ *es una expresión regular que denotal el lenguaje* $\{a^i b^j : i = 0, 1; j \geq 0\}$

Si $A = \{0, 1\}$

– $\mathbf{00}$ *es una expresión regular con lenguaje asociado* $\{00\}$.

– $\mathbf{01}^* + \mathbf{0}$ *es una expresión regular que denota el lenguaje* $\{01^i : i \geq 0\}$.

– $(\mathbf{1} + \mathbf{10})^*$ *representa el lenguaje de las cadenas que comienzan por* 1 *y no tienen dos ceros consecutivos.*

– $(\mathbf{0} + \mathbf{1})^*\mathbf{011}$ *representa el lenguaje de las cadenas que terminan en* 011.

– $\mathbf{0}^*\mathbf{1}^*$ *representa el lenguaje de las cadenas que no tienen un* 1 *antes de un* 0

– $\mathbf{00}^*\mathbf{11}^*$ *representa un subconjunto del lenguaje anterior formado por cadenas que al menos tienen un* 1 *y un* 0. *Si la expresión regular* $\mathbf{rr}^*$ *se representa como* r^+, $\mathbf{00}^*\mathbf{11}^*$ *se puede reprensentar como* $\mathbf{0}^+\mathbf{1}^+$.

2.4.1. Propiedades de las Expresiones Regulares

Dos expresiones regulares se consideran iguales si representan el mismo lenguaje. Teniendo esto en cuenta, se puede comprobar que siempre se verifican las siguientes igualdades.

1. $\mathbf{r_1} + \mathbf{r_2} = \mathbf{r_2} + \mathbf{r_1}$
2. $\mathbf{r_1} + (\mathbf{r_2} + \mathbf{r_3}) = (\mathbf{r_1} + \mathbf{r_2}) + \mathbf{r_3}$
3. $\mathbf{r_1}(\mathbf{r_2}\mathbf{r_3}) = (\mathbf{r_1}\mathbf{r_2})\mathbf{r_3}$
4. $\mathbf{r}\varepsilon = \mathbf{r}$
5. $\mathbf{r}\emptyset = \emptyset$
6. $\mathbf{r} + \emptyset = \mathbf{r}$
7. $\varepsilon^* = \varepsilon$
8. $\mathbf{r_1}(\mathbf{r_2} + \mathbf{r_3}) = \mathbf{r_1}\mathbf{r_2} + \mathbf{r_1}\mathbf{r_3}$

9. $(\mathbf{r_1}+\mathbf{r_2})\mathbf{r_3}=\mathbf{r_1}\mathbf{r_3}+\mathbf{r_2}\mathbf{r_3}$

10. $\mathbf{r}^+ + \varepsilon = \mathbf{r}^*$

11. $\mathbf{r}^* + \varepsilon = \mathbf{r}^*$

12. $(\mathbf{r}+\varepsilon)^* = \mathbf{r}^*$

13. $(\mathbf{r}+\varepsilon)^+ = \mathbf{r}^*$

14. $\left(\mathbf{r_1^*}+\mathbf{r_2^*}\right)^* = (\mathbf{r_1}+\mathbf{r_2})^*$

15. $\left(\mathbf{r_1^*}\mathbf{r_2^*}\right)^* = (\mathbf{r_1}+\mathbf{r_2})^*$

2.4.2. Expresiones Regulares y Autómatas Finitos

A continuación vamos a relacionar los lenguajes representados por las expresiones regulares con los aceptados por los autómatas finitos. Demostraremos dos teoremas. El primero nos dice que todo lenguaje asociado por una expresión regular puede ser aceptado por un autómata de estado finito. El segundo la relación recíproca. Dado un autómata de estado finito, veremos como se puede construir una expresión regular que denote el lenguaje aceptado por dicho autómata.

Teorema 4 *Dada una expresión regular existe un autómata finito que acepta el lenguaje asociado a esta expresión regular.*

Demostración.- Vamos a demostrar que existe un AFND con transiciones nulas. A partir de él se podría construir el autómata determinístico asociado.

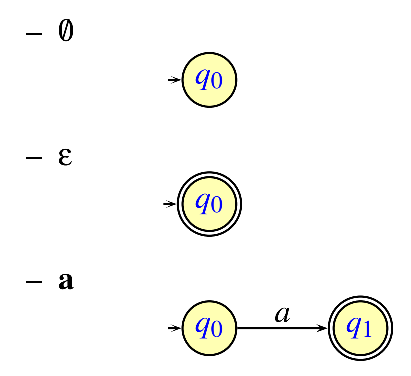
La construcción del autómata va a ser recursiva. No vamos a expresar como son los estados y las transiciones matemáticamente, sino que los autómatas vendrán expresados de forma gráfica.

Para las expresiones regulares iniciales tenemos los siguiente autómatas:

– $\emptyset$

– ε

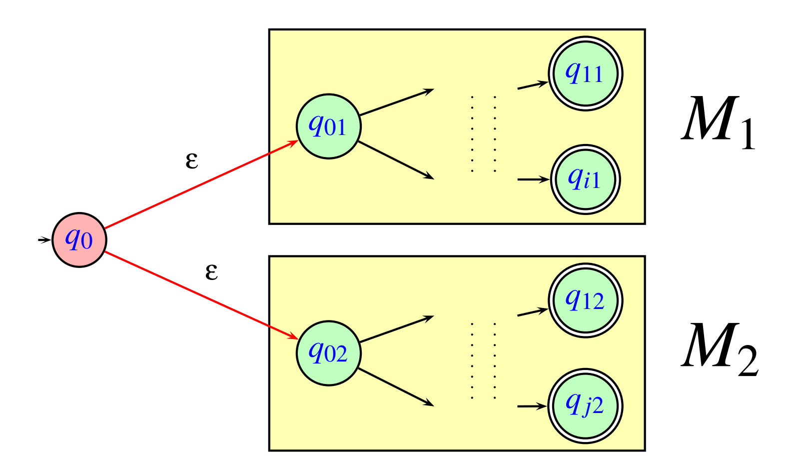
– **a**



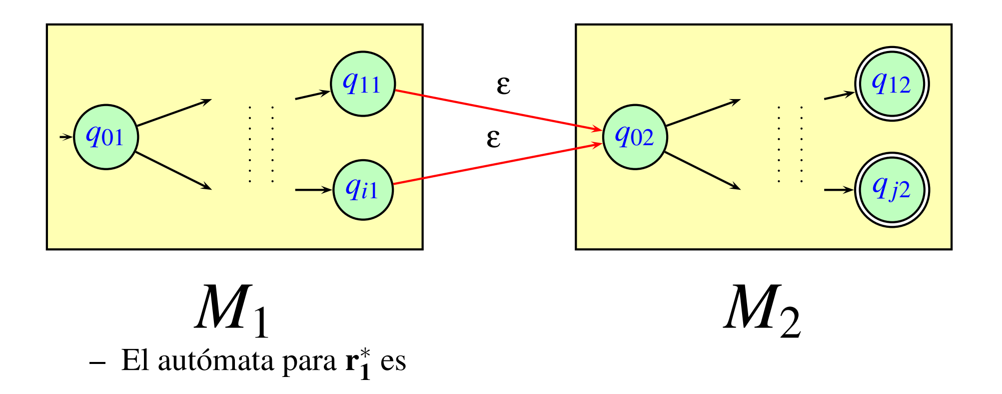
Ahora veremos como se pueden construir autómatas para las expresiones regulares compuestas a partir de los autómatas que aceptan cada una de sus componentes.

Si M_1 es el autómata que acepta el mismo lenguaje que el representado por $\mathbf{r_1}$ y M_2 el que acepta el mismo lenguaje que el de $\mathbf{r_2}$, entonces

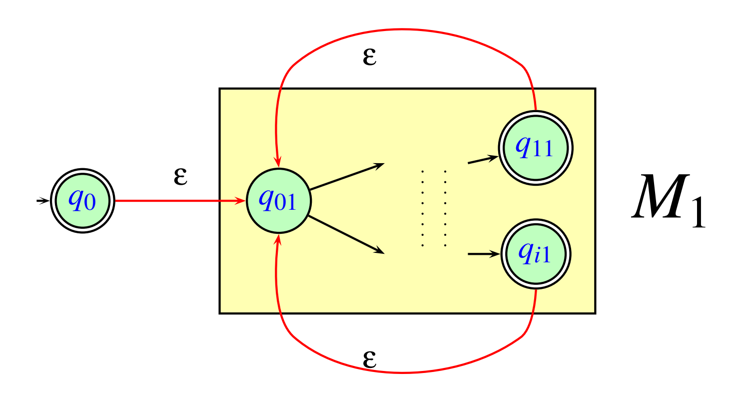
– El autómata que acepta que mismo lenguaje que el asociado a $(\mathbf{r_1} + \mathbf{r_2})$ es



– El autómata para la expresión $(\mathbf{r_1 r_2})$ es



– El autómata para $\mathbf{r_1^*}$ es

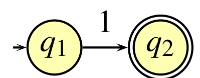


Se puede comprobar que efectivamente estos autómatas verifican las condiciones pedidas y que el procedimiento de construcción siempre acaba.

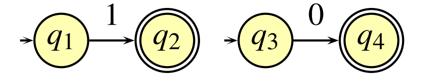
■

Ejemplo 31 *Construyamos un autómata que acepta el mismo lenguaje que el asociado a la expresión regular* $\mathbf{r} = (\mathbf{0}+\mathbf{10})^*\mathbf{011}$

– Autómata correspondiente a **1**

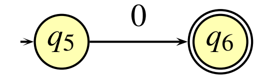


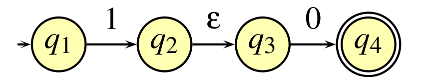
– Autómatas correspondientes a **1** y **0**



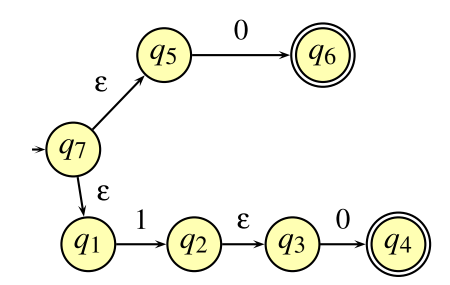
– El autómata asociado a **10** es

– Autómatas asociado a **10** y **0**

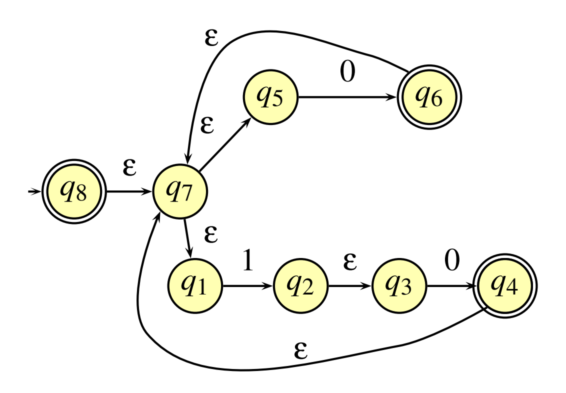




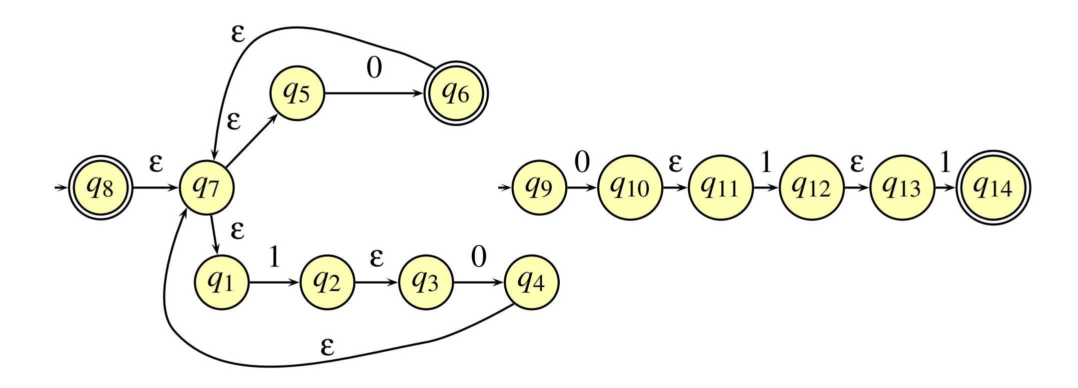
– Autómata asociado a $\mathbf{10}+\mathbf{0}$



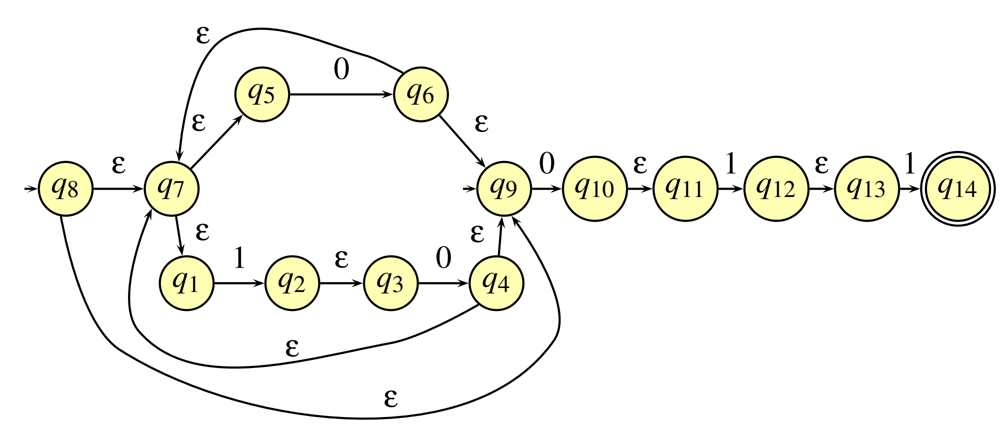
– Autómata asociado a $(\mathbf{10}+\mathbf{0})^*$



– Autómatas asociado a $(\mathbf{10}+\mathbf{0})^*$ y **011**



– Autómatas asociado a $(\mathbf{10}+\mathbf{0})^*$ y **011**



Teorema 5 *Si L es aceptado por un autómata finito determinístico, entonces puede venir expresado mediante una expresión regular.*

Demostración.-

Sea el autómata $M = (Q, A, \delta, q_1, F)$ donde $Q = \{q_1, \ldots, q_n\}$ y q_1 es el estado inicial.

Sea R_{ij}^k el conjunto de las cadenas de A^* que premiten pasar del estado q_i al estado q_j y no pasa por ningún estado intermedio de numeración mayor que k (q_i y q_j si pueden tener numeración mayor que k).

R_{ij}^k se puede definir de forma recursiva:

$$R_{ij}^0 = \begin{cases} \{a : \delta(q_i, a) = q_j\} & \text{si } i \neq j \\ \{a : \delta(q_i, a) = q_i\} \cup \{\varepsilon\} & \text{si } i = j \end{cases}$$

Para $k \geq 1$, tenemos la siguiente ecuación:

$$R_{ij}^k = R_{i(k-1)}^{k-1} (R_{(k-1)(k-1)}^{k-1})^* R_{(k-1)j}^{k-1} \cup R_{ij}^{k-1}$$

Vamos a demostrar que para todos los lenguajes R^k_{ij} existe una expresión regular $\mathbf{r^k_{ij}}$ que lo representa. Lo vamos a demostrar por inducción sobre k.

Para $k = 0$ es inmediato. La expresión regular $\mathbf{r^k_{ij}}$ puede escribirse como

$$\begin{array}{ll} \mathbf{a_1} + \ldots + \mathbf{a_l} & \text{si } i \neq j \\ \mathbf{a_1} + \ldots + \mathbf{a_l} + \varepsilon & \text{si } i = j \end{array}$$

donde $\{a_1, \ldots, a_l\}$ es el conjunto $\{a : \delta(q_i, a) = q_j\}$. Si este conjunto es vacío la expresión regular sería:

$$\begin{array}{ll} \emptyset & \text{si } i \neq j \\ \varepsilon & \text{si } i = j \end{array}$$

Es claro que estas expresiones regulares representan los conjuntos R^0_{ij}.

Supongamos ahora que es cierto para $k-1$. Entonces sabemos que

$$R^k_{ij} = R^{k-1}_{i(k-1)} \left(R^{k-1}_{(k-1)(k-1)}\right)^* R^{k-1}_{(k-1)j} \cup R^{k-1}_{ij}$$

Por la hipótesis de inducción, existen expresiones regulares para los lenguajes R^{k-1}_{lm}, que se denotan como $\mathbf{r^{k-1}_{lm}}$ De aquí se deduce que una expresión regular para R^{k-1}_{kj} viene dada por

$$\mathbf{r^{k-1}_{i(k-1)}}(\mathbf{r^{k-1}_{(k-1)(k-1)}})^* \mathbf{r^{k-1}_{(k-1)j}} + \mathbf{r^{k-1}_{ij}}$$

con lo que concluye la demostración de que existen expresiones regulares para los lenguajes R^k_{ij} . Además esta demostración nos proporciona un método recursivo para calcular estas expresiones regulares.

Finalmente, para demostrar que el lenguaje aceptado por el autómata puede venir expresado mediante una expresión regular, solo hay que observar que

$$L(M) = \bigcup_{q_j \in F} R^n_{1j}$$

Por tanto, $L(M)$ viene denotado por la expresión regular $\mathbf{r_{j_1}} + \ldots + \mathbf{r_{j_k}}$ donde $F = \{q_{j_1}, \ldots, q_{j_k}\}$.

■

Ejemplo 32 *Sea el Autómata Finito Determinístico de la Figura 32. Vamos a construir la expresión regular que representa el lenguaje aceptado por este autómata.*

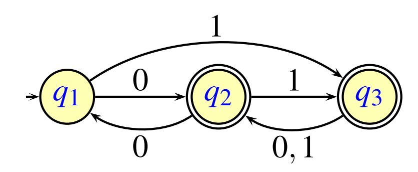


Figura 2.12: Autómata Finito Determinístico

$r^0_{11} = \varepsilon$
$r^0_{12} = 0$
$r^0_{13} = 1$
$r^0_{21} = 0$
$r^0_{22} = \varepsilon$
$r^0_{23} = 1$
$r^0_{31} = \emptyset$
$r^0_{32} = 0+1$
$r^0_{33} = \varepsilon$
$r^1_{11} = r^0_{11} + r^0_{11}(r^0_{11})^* r^0_{11} = \varepsilon + \varepsilon(\varepsilon)^*\varepsilon = \varepsilon$
$r^1_{12} = r^0_{12} + r^0_{11}(r^0_{11})^* r^0_{12} = 0 + \varepsilon(\varepsilon)^* 0 = 0$
$r^1_{13} = r^0_{13} + r^0_{11}(r^0_{11})^* r^0_{13} = 1 + \varepsilon(\varepsilon)^* 1 = 1$
$r^1_{21} = r^0_{21} + r^0_{21}(r^0_{11})^* r^0_{11} = 0 + 0(\varepsilon)^*\varepsilon = 0$
$r^1_{22} = r^0_{22} + r^0_{21}(r^0_{11})^* r^0_{12} = \varepsilon + 0(\varepsilon)^* 0 = \varepsilon + 00$
$r^1_{23} = r^0_{23} + r^0_{21}(r^0_{11})^* r^0_{13} = 1 + 0(\varepsilon)^* 1 = 1 + 01$
$r^1_{31} = r^0_{31} + r^0_{31}(r^0_{11})^* r^0_{11} = \emptyset + \emptyset(\varepsilon)^*\varepsilon = \emptyset$
$r^1_{32} = r^0_{32} + r^0_{31}(r^0_{11})^* r^0_{12} = 0 + 1 + \emptyset(\varepsilon)^* 0 = 0 + 1$
$r^1_{33} = r^0_{33} + r^0_{31}(r^0_{11})^* r^0_{13} = \varepsilon + \emptyset(\varepsilon)^* 1 = \varepsilon$
$r^2_{11} = r^1_{11} + r^1_{12}(r^1_{22})^* r^1_{21} = \varepsilon + 0(\varepsilon + 00)^* 0 = (00)^*$
$r^2_{12} = r^1_{12} + r^1_{12}(r^1_{22})^* r^1_{22} = 0 + 0(\varepsilon + 00)^*(\varepsilon + 00) = 0(00)^*$
$r^2_{13} = r^1_{13} + r^1_{12}(r^1_{22})^* r^1_{23} = 1 + 0(\varepsilon + 00)^*(1 + 01) = 0^* 1$
$r^2_{21} = r^1_{21} + r^1_{22}(r^1_{22})^* r^1_{21} = 0 + (\varepsilon + 00)(\varepsilon + 00)^* 0 = (00)^* 0$
$r^2_{22} = r^1_{22} + r^1_{22}(r^1_{22})^* r^1_{22} = \varepsilon + 00 + (\varepsilon + 00)(\varepsilon + 00)^*(\varepsilon + 00) = (00)^*$
$r^2_{23} = r^1_{23} + r^1_{22}(r^1_{22})^* r^1_{23} = 1 + 01 + (\varepsilon + 00)(\varepsilon + 00)^*(1 + 01) = 0^* 1$
$r^2_{31} = r^1_{31} + r^1_{32}(r^1_{22})^* r^1_{21} = \emptyset + (0+1)(\varepsilon + 00)^* 0 = (0+1)(00)^* 0$
$r^2_{32} = r^1_{32} + r^1_{32}(r^1_{22})^* r^1_{22} = 0 + 1 + (0+1)(\varepsilon + 00)^*(\varepsilon + 00) = (0+1)(00)^*$
$r^2_{33} = r^1_{33} + r^1_{32}(r^1_{22})^* r^1_{23} = \varepsilon + (0+1)(\varepsilon + 00)^*(1 + 01) = \varepsilon + (0+1)0^* 1$

Finalmente la expresión regular para el lenguaje aceptado es:

$$r^3_{12} + r^3_{13} = r^2_{12} + r^2_{13}(r^2_{33})^* r^2_{32} + r^2_{13} + r^2_{13}(r^2_{33})^* r^2_{33} =$$

$$0(00)^* + 0^*1(\varepsilon + (0+1)0^*1)^*(0+1)(00)^* + 0^*1 + 0^*1(\varepsilon + (0+1)0^*1)^*(\varepsilon + (0+1)0^*1) =$$

$$0(00)^* + 0^*1((0+1)0^*1)^*(0+1)(00)^* + 0^*1 + 0^*1((0+1)0^*1)^*$$

Por último señalaremos que, por abuso del lenguaje, se suele identificar una expresión regular con el lenguaje asociado. Es decir, diremos el lenguaje $(\mathbf{00}+\mathbf{11})^*$, cuando, en realidad, nos estaremos refiriendo al lenguaje asociado a esta expresión regular.

2.5. Gramáticas Regulares

Estas son las gramáticas de tipo 3. Pueden ser de dos formas

– *Lineales por la derecha.*- Cuando todas las producciones tienen la forma

$$A \to uB$$
$$A \to u$$

– *Lineales por la izquierda.*- Cuando todas las producciones tienen la forma

$$A \to Bu$$
$$A \to u$$

Ejemplo 33 *La gramática dada por* $V = \{S, A\}, T = \{0, 1\}$ *y las producciones*

$$S \to 0A$$
$$A \to 10A$$
$$A \to \varepsilon$$

es lineal por la derecha. Genera el lenguaje $\mathbf{0}(\mathbf{01})^*$ *.*

El mismo lenguaje es generado por la siguiente gramática lineal por la izquierda

$$S \to S10$$

$$S \to 0$$

Teorema 6 *Si L es un lenguaje generado por una gramática regular, entonces existe un autómata finito determinístico que lo reconoce.*

Demostración.- Supongamos que L es un lenguaje generado por la gramática $G = (V,T,P,S)$ que es lineal por la derecha. Vamos a construir un AFND con movimientos nulos que acepta L.

Este autómata será $M = (Q,T,\delta,q,F)$ donde

- $Q = \{[\alpha] : (\alpha = S) \vee (\exists A \in V, u \in T, \text{ tales que } A \rightarrow u\alpha \in P)\}$
- $q_0 = [S]$
- $F = \{[\varepsilon]\}$
- δ viene definida por
  - Si A es una variable
$$\delta([A],\varepsilon) = \{[\alpha] : (A \rightarrow \alpha) \in P\}$$
  - Si $a \in T$ y $\alpha \in (T^*V)$, entonces
$$\delta([a\alpha],a) = [\alpha]$$

La aceptación de una palabra en este autómata simula la aplicación de reglas de derivación en la gramática original. La demostración formal de que esto es así no la vamos a considerar.

En el caso de una gramática lineal por la izquierda, $G = (V,T,P,S)$, consideraremos la gramática $G' = (V,T,P',S)$ donde $P' = \{A \rightarrow \alpha : A \rightarrow \alpha \in P\}$. Es decir que invertimos la parte derecha de las producciones. La gramática resultante G' es lineal por la derecha y el lenguaje que genera es $L(G') = L(G)^{-1}$.

Ahora, podemos construír un autómata que acepte el lenguaje $L(G')$, siguiendo el procedimiento anterior. El autómata no determinístico correspondiente se puede transformar en uno equivalente con un solo estado final. Para ello basta con añadir un nuevo estado final, pasar a no-finales los estados finales originales y unir estos mediante una transición nula con el nuevo estado final.

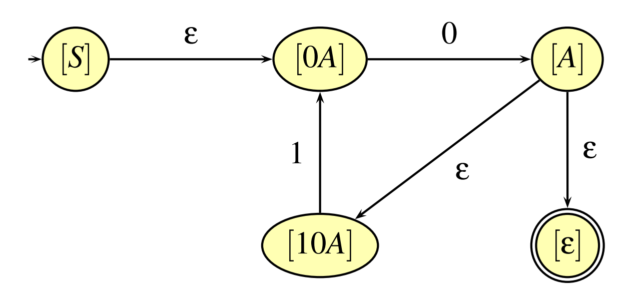
El siguiente paso es invertir el autómata ya con un solo estado final, para que pase de aceptar el lenguaje $L(G') = L(G)^{-1}$ al lenguaje $L(G')^{-1} = L(G)$. Para ello los pasos son:

- Invertir las transiciones
- Intercambiar el estado inicial y el final.

A partir de este autómata se podría construir un autómata determinístico, con lo que termina la demostración.

■

Ejemplo 34 *De la gramática* $S \rightarrow 0A, A \rightarrow 10A, A \rightarrow \varepsilon$ *se obtiene el autómata,*



Ejemplo 35 *La gramática lineal por la izquierda que genera el lenguaje* $0(01)^*$ *tiene las siguientes producciones*

$S \rightarrow S10$

$S \rightarrow 0$

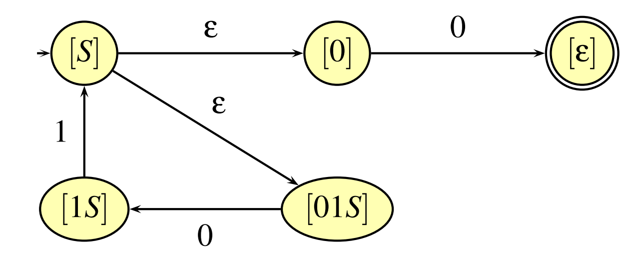
Para construir un AFND con transiciones nulas qque acepte este lenguaje se dan los siguientes pasos:

1. *Invertir la parte derecha de las producciones*

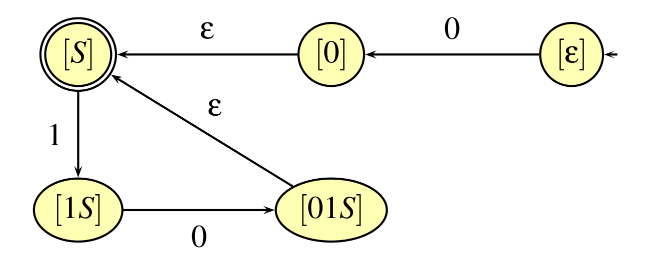
   $S \rightarrow 01S$

   $S \rightarrow 0$

2. *Construir el AFND con transiciones nulas asociado*



3. *Invertimos las transiciones*



Teorema 7 *Si L es aceptado por un Autómata Finito Determinístico entonces L puede generarse mediante una gramática lineal por la derecha y por una lineal por la izquierda.*

Demostración.- Sea $L = L(M)$ donde $M = (Q, A, \delta, q, F)$ es un autómata finito determinístico. Construiremos, en primer lugar, una gramática lineal por la derecha.

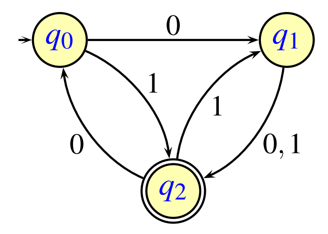
La gramática que construimos es $G = (Q, A, P, q_0)$ donde las variables son los estados, la variable inicial es q_0 y P contiene las producciones,

$$p \rightarrow aq, \quad \text{si } \delta(p,a) = q$$

$$p \rightarrow \varepsilon, \quad \text{si } p \in F$$

Para el caso de una gramática lineal por la izquierda, invertimos el autómata, construímos la gramática lineal por la derecha asociada e invertimos la parte derecha de las producciones. ■

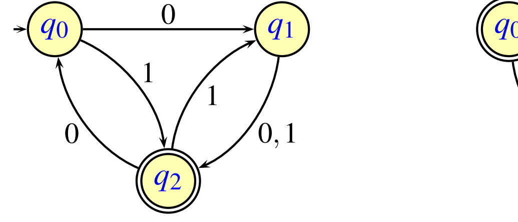
Ejemplo 36 *Consideremos el autómata:*



La gramática lineal por la derecha es (variable inicial q_0):

$$q_0 \rightarrow 0q_1, \quad q_0 \rightarrow 1q_2, \quad q_1 \rightarrow 0q_2, \quad q_1 \rightarrow 1q_2$$
$$q_2 \rightarrow 0q_0, \quad q_2 \rightarrow 1q_1, \quad q_2 \rightarrow \varepsilon$$

Si lo que queremos es una gramática lineal por la izquierda, entonces invertimos el autómata:





La gramática asociada es (variable inicial q_2):

$$q_1 \rightarrow 0q_0, \quad q_2 \rightarrow 1q_0, \quad q_2 \rightarrow 0q_1, \quad q_2 \rightarrow 1q_1$$
$$q_0 \rightarrow 0q_2, \quad q_1 \rightarrow 1q_2, \quad q_0 \rightarrow \varepsilon$$

Invertimos la parte derecha de las producciones, para obtener la gramática lineal por la izquierda asociada:

$$q_1 \rightarrow q_0 0, \quad q_2 \rightarrow q_0 1, \quad q_2 \rightarrow q_1 0, \quad q_2 \rightarrow q_1 1$$
$$q_0 \rightarrow q_2 0, \quad q_1 \rightarrow q_2 1, \quad q_0 \rightarrow \varepsilon$$

2.6. Máquinas de Estado Finito

Las máquinas de estado finito son autómatas finitos con salida. Veremos dos tipos de máquinas:

- M áquinas de Moore: con salida asociada al estado
- M áquinas de Mealy: con salida asociada a la transición

Sin embargo, ambas máquinas son equivalentes, en el sentido de que calculan las mismas funciones.

2.6.1. Máquinas de Moore

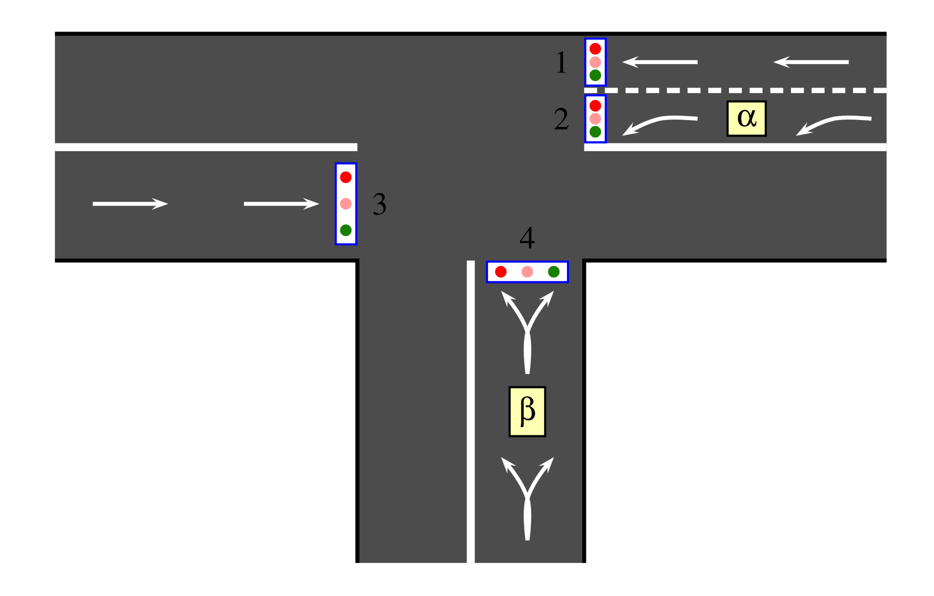
Una máquina de Moore es una sextupla
$\{(Q, A, B, \delta, \lambda, q_0)\}$ donde todos los elementos son como en los autómatas finitos determinísticos, excepto

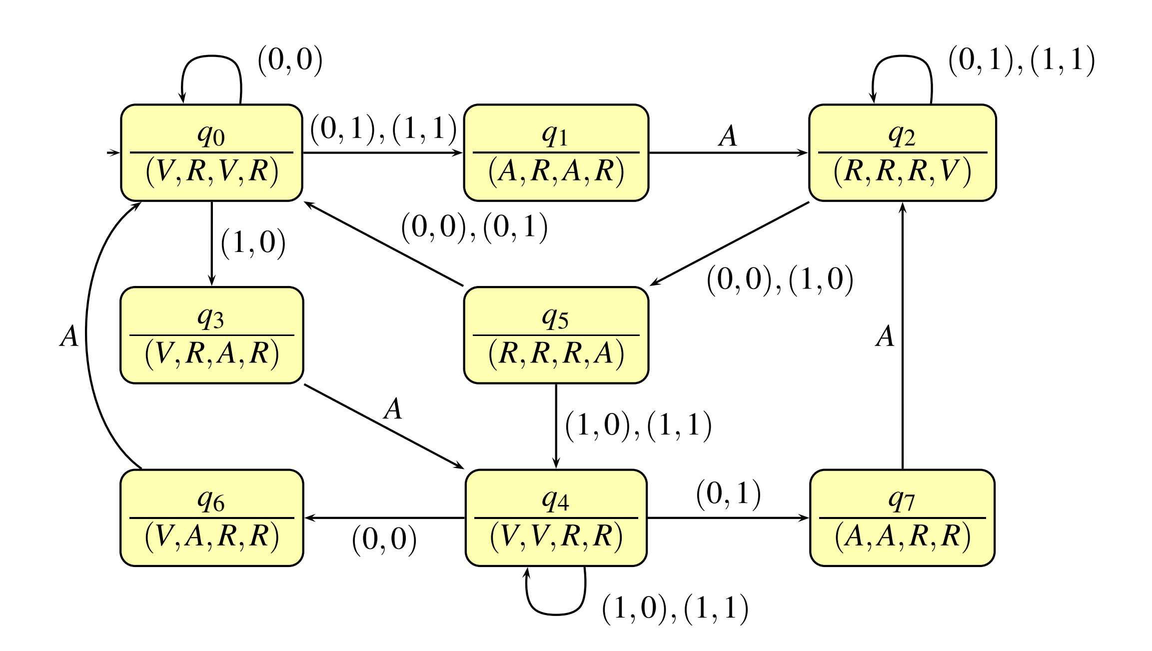
- B alfabeto de salida
- $\lambda : Q \rightarrow B$ que es una aplicación que hace corresponder a cada estado su salida correspondiente.

No hay estados finales, porque ahora la respuesta no es 'acepta' o 'no acepta' sino que es una cadena formada por los símbolos de salida correspondientes a los estados por los que pasa el autómata.

Si el autómata lee la cadena u y pasa por los estados $q_0q_1...q_n$ entonces produce la salida $\lambda(q_0)\lambda(q_1)\ldots\lambda(q_n)$

Es conveniente señalar que produce una salida para la cadena vacía: $\lambda(q_0)$.


1
2
α
3
4
β


(0,0)
(0,1),(1,1)
q_0
(V,R,V,R)
(0,1),(1,1)
q_1
(A,R,A,R)
A
q_2
(R,R,R,V)
(1,0)
(0,0),(0,1)
(0,0),(1,0)
A
q_3
(V,R,A,R)
q_5
(R,R,R,A)
A
A
(1,0),(1,1)
q_6
(V,A,R,R)
(0,0)
q_4
(V,V,R,R)
(0,1)
q_7
(A,A,R,R)
(1,0),(1,1)

Ejemplo 37 *Control de semáforos en un cruce. Consideremos un cruce de carreteras, cuya estructura viene dada por la figura 37*

El tráfico importante va a estar en la carretera horizontal. La otra dirección es menos densa. Por eso se han puesto dos sensores, α *y* β*, que mandan información sobre si hay coches esperando en las colas de los sem áforos 2 y 4 respectivamente: 1 si hay coches esperando y 0 si no hay coches. Estos semáforos solo se abrirán en el caso de que hay coches esperando en la cola. En caso contrario permanecer n cerrados.*

Este cruce va a ser controlado por una Máquina de Moore que lee los datos de los sensores: pares (a,b) *donde a es la información del sensor* α *y b la información del sensor* β*. Las salidas será n cuadruplas* (a_1,a_2,a_3,a_4) *donde* $a_i \in \{R,A,V\}$ *indicando como se colocan los sem foros.*

El esquema de una máquina de Moore que controla el tráfico es el de la figura anterior. Junto a cada estado se especifica la salida correspondiente a dicho estado.

2.6.2. Máquinas de Mealy

Una Máquina de Mealy es también una sextupla $M = (Q,A,B,\delta,\lambda,q_0)$ donde todo es igual que en las máquinas de Moore, excepto que λ es una aplicación de $Q \times A$ en B,

$\lambda : Q \times A \rightarrow B$

es decir, que la salida depende del estado en el que está el autómata y del símbolo leido.

Si la entrada es $a_1 \dots a_n$ y pasa por los estados $q_0,q_1,\dots,q_n$, la salida es

$$\lambda(q_0,a_1)\lambda(q_1,a_2)\dots\lambda(q_{n-1},a_n)$$

Si se le suministra ε como entrada produce ε como salida.

Ejemplo 38 *Supongamos una máquina codificadora que actúa de la siguiente forma:*

El alfabeto de entrada es $\{0,1\}$ *y el de salida* $\{0,1\}$*. La traducción viene dada por las siguientes reglas,*

- *Primer símbolo*

$$0 \rightarrow 0$$

$$1 \rightarrow 1$$

- *Siguientes símbolos*

  • *Si el anterior es un 0*

$$0 \rightarrow 0$$

$$1 \rightarrow 1$$

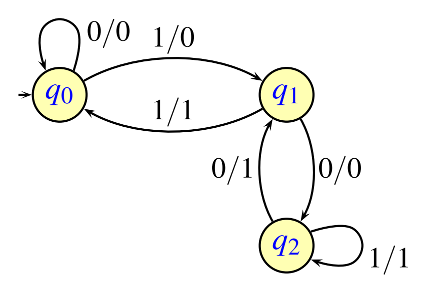


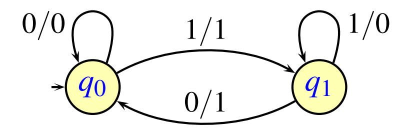
Figura 2.13: Máquina de Mealy que realiza la división entera por 3

- *Si el anterior es un 1*

$$0 \rightarrow 1$$

$$1 \rightarrow 0$$

La máquina de Mealy que realiza esta codificación es la siguiente



si lee 0101, la salida correspondiente es 0111.

Ejemplo 39 *Una máquina de Mealy que realiza la división entera por 3 es el de la figura 2.13.*

2.6.3. Equivalencia de Máquinas de Mealy y Máquinas de Moore

Una máquina de Mealy y una máquina de Moore calculan funciones aná logas. Sea M una máquina de Moore y M' una máquina de Mealy. Si notamos como $T_M(u) y T'_M(u)$ las salidas que producen ante una entrada u, entonces se puede comprobar que siempre, $|T_M(u)| = |T'_M(u)| + 1$. Luego, en sentido estricto, nunca pueden ser iguales las funciones que calculan una máquina de Mealy y una máquina de Moore. Sin embargo, la primera salida de una máquina de Moore es siempre la misma: la correspondiente al estado inicial. Si despreciamos esta salida, que es siempre la misma, entonces si podemos comparar las funciones calculadas por las máquinas de Mealy y de Moore.

Definición 39 *Una máquina de Moore, M, y una máquina de Mealy, M', se dicen equivalentes sii paratodo* $u \in A^*$ $T_M(u) = bT_{M'}(u)$ *donde b es la salida correspondiente al estado inicial de la máquina de Moore M.*

Teorema 8 *Dada una máquina de Moore, existe una máquina de Mealy equivalente.*

Demostración.- Sea $M = (Q, A, B, \delta, \lambda, q_0)$ una máquina de Moore, la máquina de Mealy equivalente será $M' = (Q, A, B, \delta, \lambda', q_0)$, donde

$$\lambda'(q, a) = \lambda(\delta(q, a))$$

Es decir se le asigna a cada transición la salida del estado de llegada en la máquina de Moore. Es inmediato comprobar que ambas máquinas son equivalentes.

■

Teorema 9 *Dada una máquina de Mealy, existe una máquina de Moore equivalente*

Demostración.- Sea $M = (Q, A, B, \delta, \lambda, q_0)$ una máquina de Mealy.
La máquina de Moore será: $M = (Q', A, B, \delta', \lambda', q'_0)$ donde

- $Q' = Q \times B$
- $\delta'((q, b), a) = (\delta(q, a), \lambda(q, a))$
- $\lambda'(q, b) = b$
- $q'_0 = (q_0, b)$, donde $b \in B$, cualquiera.

■

Ejemplo 40 *Dada la máquina de Mealy de la figura 2.13, aplicando el procedimiento del teorema anterior, obtenemos la máquina de Moore equivalente de la figura 2.14.*

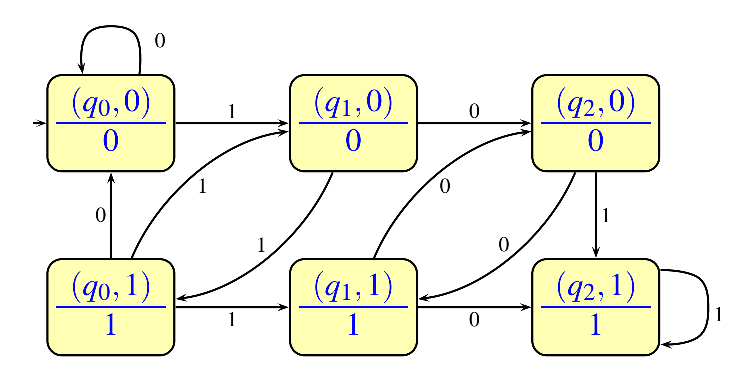


Figura 2.14: Máquina de Moore que realiza la división entera por 3

Capítulo 3

Propiedades de los Conjuntos Regulares

3.1. Lema de Bombeo

A los lenguajes aceptados por un AFD se les llama también **conjuntos regulares**. El lema de bombeo se usa para demostrar que un determinado conjunto no es regular, es decir, no puede llegar a ser aceptado por un autómata finito determinístico.

Lema 1 (Lema de Bombeo) *Sea L un conjunto regular, entonces existe un $n \in N$ tal que $\forall z \in L$, si $|z| \geq n$, entonces z se puede expresar de la forma $z = uvw$ donde*

1. $|uv| \leq n$
2. $|v| \geq 1$
3. $(\forall i \geq 0) uv^i w \in L$

además n puede ser el número de estados de cualquier autómata que acepte el lenguaje L.

Demostración.-

Sea M un autómata finito determinístico que acepta el lenguaje L.

Supongamos que el conjunto de estados es $Q = \{q_0, q_1, \ldots, q_k\}$.

Hagamos $n = k+1$.

Entonces si $z \in L$ y $|z| \geq n = k+1$ resulta que z tiene, al menos, tantos símbolos como estados tiene el autómata.

Cuando el autómata lee la palabra z, realiza un cambio de estado por cada símbolo de dicha palabra, empezando en q_0 y terminando en un estado final q_m .

Sea z' la parte de z constituida por los n primeros símbolos de z. Consideremos el vector de los estados por los que $(q_{i_0}, q_{i_1}, \ldots, q_{i_n})$ donde q_{i_0} es el estado inicial, q_0, y $q_{i_j} = \delta(q_{i_{j-1}}, a_j)$.

Como el vector $(q_{i_0}, q_{i_1}, \ldots, q_{i_n})$ tiene $n+1$ estados y sólo hay n estados distintos, tiene que haber necesariamente un estado que se repita. Sea q_l el primer estado que se repite en este vector. Entonces consideramos:

- Sea u la parte de la palabra z' que lleva al autómata desde q_0 a la primera aparición de q_l.
- Sea v la parte de la palabra de z', que lleva el autómata de la primera aparición de q_l a la segunda aparición de q_l.
- Sea w lo que le falta a uv para completar la palabra z.

Con esta partición se puede probar:

1. $z = uvw$

2. $|uv| \leq n$, porque uv forman parte de z' que sólo tiene n símbolos.

3. $|v| \geq 1$, porque para pasar de la primera aparición de q_l a la segunda, hay que leer al menos un símbolo.

4. $uv^iw \in L, \forall i \in N$. En efecto, esta palabra es aceptada por el autómata, porque u lleva al autómata del estado inicial q_0 al estado q_l. Cada aparición de v mantiene al autómata en el mismo estado q_l . Por último, w lleva al autómata desde q_l al estado final q_m. Como después de leer uv^iw llegamos a un estado final, entonces la palabra es aceptada por el autómata y pertenece al lenguaje L.

Con esto queda demostrado el lema. ■

Para demostrar que un determinado lenguaje no es regular, basta probar que no se verifica la condición que aparece en el lema de bombeo. Para ello hay que dar los siguiente pasos:

1. Suponer un $n \in N$ arbitrario que se suponga cumpla las condiciones del lema.

2. Encontrar una palabra, z (que puede depender de n) de longitud mayor o igual que n para la que sea imposible encontrar una partición como la del lema.

3. Demostrar que una palabra z no cumple las condiciones del lema, para lo que hay que hacer:

   a) Suponer una partición arbitraria de z, $z = uvw$, tal que $|uv| \leq n$, $|v| \geq 1$.

   b) Encontrar un $i \in N$ tal que $uv^iw \notin L$.

Ejemplo 41 *Demostrar que el lenguaje* $L = \{0^k1^k : k \geq 0\}$ *no es regular.*

1. Supongamos un n cualquiera

2. Sea la palabra 0^n1^n *que es de longitud mayor o igual que n*

3. Demostremos que 0^n1^n *no se puede descomponer de acuerdo con las condiciones del lema:*

   a*)* *Supongamos una partición arbitraria de* $0^n1^n = uvw$. *con* $|uv| \leq n$ *y* $|v| \geq 1$.
   Como $|uv| \leq n$ *resulta que v está formada de* 0 *solamente y como* $|v| \geq 1$, *v tiene, al menos un cero.*

b) *Basta considerar* $i = 2$ *y*
$uv^2w = uvvw = 0^n v 1^n$ *y esta no es una palabra de L, ya que si v tiene solo ceros, no hay el mismo número de ceros que de unos, y si v tiene ceros y unos, z tendrá unos antes de algún cero.*

Con esto es suficiente para probar que L no es regular. Un resultado mucho mas difícil de encontrar directamente.

Ejemplo 42 *Demostrar que el lenguaje* $L = \{0^{j^2} : j \geq 0\}$ *no es regular.*
Sea un $n \in N$*, consideremos la palabra* $z = 0^{n^2} \in L$ *y de longitud mayor o igual a n.*
Supongamos una descomposición cualquiera $z = uvw$*. Si se verifica*

- $|uv| \leq n$
- $|v| \geq 1$

entonces tenemos que $u = 0^k, v = 0^l, w = 0^{n^2-l-k}$*, con* $l \geq 1, l \leq n$*.*
Haciendo $i = 2, uv^2w = 0^k 0^{2l} 0^{n^2-l-k} = 0^{n^2+l}$*.*
Como $(n+1)^2 - n^2 = n^2 + 2n + 1 - n^2 = 2n + 1 > n \geq l$*, tenemos que* $n^2 < n^2 + l < (n+1)^2$ *y* $uv^2w = 0^{n^2+l} \notin L$*.*
Con esto el lenguaje no puede ser regular.

Ejemplo 43 *El lenguaje* $\{u \in \{0,1\}^* : u = u^{-1}\}$ *no es regular.*
Sea un $\forall n \in N$ *cualquiera. Consideremos la palabra* $z = 0^n 1 0^n \in L$ *y que tiene longitud* $2n + 1 \geq n$*.*
Supongamos una descomposición cualquiera: $z = uvw$*.*
Si se verifica

- $|uv| \leq n$
- $|v| \geq 1$

entonces tenemos que $u = 0^k, v = 0^l, w = 0^{n-k-l} 1^n 0^n$*, con* $l \geq 1$*.*
Haciendo $i = 2, uv^2w = 0^k 0^{2l} 0^{n-k-l} 1^n 0^n = 0^{n+l} 1^n 0^n \notin L$*.*
Como hemos encontrado un i tal que $uv^iw \notin L$*, tenemos que el lenguaje no verifica la condición que aparece en el lema de bombeo y no puede ser regular.*

Hay lenguajes que no son regulares y sin embargo verifican la condición que aparece en el lema de bombeo. A continuación se da un ejemplo de este caso.

Ejemplo 44 *Sea* $L = \{a^i b^j c^k : (i = 0) \vee (j = k)\}$ *un lenguaje sobre el alfabeto* $A = \{a, b, c\}$*. Vamos a demostrar que se verifica la condición del lema de bombeo para* $n = 2$*.*
En efecto si $z \in L$ *y* $|z| \geq 2$ *entonces* $z = a^i b^j z^k$ *con* $i = 0$ *o* $j = k$*. Caben dos posibilidades:*

a) $i = 0$. *En este caso* $z = b^j c^k$ *una descomposición de* z *se puede obtener de la siguiente forma:*

$u = \varepsilon$

v *es el primer símbolo de* z

w *es* z *menos su primer símbolo*

Está claro que se verifican las tres condiciones exigidas en el lema de bombeo.

1. $|uv| = 1 \leq n = 2$
2. $|v| = 1 \geq 0$
3. *Si* $l \geq 0$ *entonces* $uv^l w$ *sigue siendo una sucesión de* b *seguida de una sucesión de* c *y por tanto una palabra de* L.

b) $i \geq 0$. *En ese caso* $z = ab^j c^j$, *y una partición de esta palabra se obtiene de acuerdo con lo siguiente:*

$u = \varepsilon$

$v = a$ *es el primer símbolo de* z

w *es* z *menos su primer símbolo*

También aquí se verifican las tres condiciones:

1. $|uv| = 1 \leq n = 2$
2. $|v| = 1 \geq 0$
3. *Si* $l \geq 0$ *entonces* $uv^l w$ *sigue siendo una sucesión de* a *seguida de una sucesión de* b *y otra de* c, *en la que la cantidad de* b *es igual que la cantidad de* c, *y por tanto, una palabra de* L.

Es fácil intuir que este lenguaje no es regular, ya que el número de estados necesarios para tener en cuenta el número de a *y de* b *puede hacerse ilimitado (y no sería finito).*

3.2. Operaciones con Conjuntos Regulares

Ya conocemos las siguientes propiedades,

- *Unión:* Si L_1 y L_2 son conjuntos regulares, entonces $L_1 \cup L_2$ es regular.
- *Concatenación:* Si L_1 y L_2 son regulares, entonces $L_1 L_2$ es regular.
- *Clausura de Kleene:* Si L es regular, entonces L^* es regular.

Adicionalmente, vamos a demostrar las siguientes propiedades.

Proposición 1 *Si $L \subseteq A^*$ es un lenguaje regular entonces $\overline{L} = A^* - L$ es regular.*

Demostración.- Basta con considerar que si $M = (Q, A, \delta, q_0, F)$ es un autómata finito determinístico que acepta el lenguaje L, entonces $M' = (Q, A, \delta, q_0, Q - F)$ acepta el lenguaje complementario $A^* - L$. ■

Proposición 2 *Si L_1 y L_2 son dos lenguajes regulares sobre el alfabeto A, entonces $L_1 \cap L_2$ es regular.*

Demostración.-

Es inmediato ya que $L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}$.

Existe también una demostración constructiva. Si $M_1 = (Q_1, A, \delta_1, q_0^1, F_1)$ es un autómata finito determinístico que acepta L_1, y $M_2 = (Q_2, A, \delta_2, q_0^2, F_2)$ es un autómata que acepta L_2, entonces

$M = (Q_1 \times Q_2, A, \delta, (q_0^1, q_0^2), F_1 \times F_2)$

donde $\delta((q_i, q_j), a) = (\delta(q_i, a), \delta(q_j, a))$, acepta el lenguaje $L_1 \cap L_2$. ■

Proposición 3 *Si A y B son alfabetos y $f : A^* \longrightarrow B^*$ un homomorfismo entre ellos, entonces si $L \subseteq A^*$ es un lenguaje regular, $f(L) = \{u \in B : \exists v \in A$ verificando $f(v) = u\}$ es también un lenguaje regular.*

Demostración.-

Basta con comprobar que se puede conseguir una expresión regular para $f(L)$ partiendo de una expresión regular para L: Basta con substituir cada símbolo, a, de L, por la correspondiente palabra $f(a)$. ■

Ejemplo 45 *Si $A = \{0,1,2,3,4,5,6,7,8,9\}$ y $B = \{0,1\}$ y f es el homomorfismo dado por*
$f(0) = 0000, \quad f(1) = 0001, \quad f(2) = 0010, \quad f(3) = 0011$
$f(4) = 0100, \quad f(5) = 0101, \quad f(6) = 0110, \quad f(7) = 0111$
$f(8) = 1000, \quad f(9) = 1001$

Entonces si $L \subseteq A^*$ es el lenguaje regular dado por la expresión regular $(1+2)^*9$, entonces el lenguaje $f(L)$ también es regular y viene dado por la expresión regular $(0001+0010)^*1001$.

Proposición 4 *Si A y B son alfabetos y $f : A^* \longrightarrow B^*$ es un homomorfismo, entonces si $L \subseteq B^*$ es un conjunto regular, también lo es $f^{-1}(L) = \{u \in A : f(u) \in L\}$.*

Demostración.-

Supongamos que $M = (Q,B,\delta,q_0,F)$ es un autómata que acepta el lenguaje L, entonces el autómata $M = (Q,A,\overline{\delta},q_0,F)$ donde

$$\overline{\delta}(q,a) = \delta'(q,f(a)),$$

acepta el lenguaje $f^{-1}(L)$. ■

Ejemplo 46 *Si $A = B = \{0,1\}$ y f es el homomorfismo dado por*

$$f(0) = 00, \qquad f(1) = 11$$

entonces el lenguaje $L = \{0^{2k}1^{2k} : k \geq 0\}$ no es regular, porque si lo fuese su imagen inversa, $f^{-1}(L) = \{0^k1^k : k \geq 0\}$ sería también regular y no lo es.

Proposición 5 *Si R es un conjunto regular y L un lenguaje culaquiera, entonces el cociente de lenguajes $R/L = \{u : \exists v \in L$ verificando $uv \in R\}$ es un conjunto regular.*

Demostración.-

Sea $M = (Q,A,\delta,q_0,F)$ un autómata finito determinístico que acepta el lenguaje R.

Entonces R/L es aceptado por el autómata

$$M' = (Q,A,\delta,q_0,F')$$

donde $F' = \{q \in Q : \exists y \in L$ tal que $\delta'(q,y) \in F\}$

Esta demostración no es constructiva, en el sentido de que puede que no exista un algoritmo tal que dado un estado $q \in Q$ nos diga si existe una palabra $y \in L$ tal que $\delta(q,y) \in L$. Si L es regular, se puede asegurar que dicho algoritmo existe.

■

3.3. Algoritmos de Decision para Autómatas Finitos

El lema de bombeo sirve para encontrar algoritmos de decisión para los autómatas finitos, por ejemplo, para saber si el lenguaje que acepta un determinado autómata es vacío o no.

Teorema 10 *El conjunto de palabras aceptado por un autómata finito con n estados es*

1. *No vacío si y solo si existe una palabra de longitud menor que n que es aceptada por el autómata.*
2. *Infinito si y solo si acepta una palabra de longitud l, donde $n \leq l < 2n$.*

Demostración.-

1. La parte *si* es evidente: si acepta una palabra de longitud menor o igual que n, el lenguaje que acepta no es vacío.

   Para la parte *solo si*, supongamos que el lenguaje aceptado por un autómata es no vacío. Consideremos una palabra, u, de longitud mínima aceptada por el autómata. En dicho caso, $|u| < n$, porque si $|u| \geq n$, entonces por el lema de bombeo $u = xyz$, donde $|y| \geq 1$ y xz es aceptada por el autómata, siendo $|xz| < |u|$, en contra de que u sea de longitud mínima.

2. Si $u \in L(M)$ y $n \leq |u| < 2n$, entonces por el lema de bombeo se pueden obtener infinitas palabras aceptadas por el autómata y $L(M)$ es infinito.

   Reciprocamente, si $L(M)$ es infinito, habrá infinitas palabras de longitud mayor o igual que n. Sea u una palabra de longitud mínima de todas aquellas que tienen de longitud mayor o igual que n.

   Entonces, u ha de tener una longitud menor de $2n$, porque si no es así, por el lema

   de bombeo, u se puede descomponer $u = xyz$, donde $1 \leq |y| \leq n$, y $xz \in L(M)$. Por tanto, xz sería aceptada por el autómata, tendría una longitud mayor o igual que n, y tendría menos longitud que u. En contra de que u sea de longitud mínima.

■

Este teorema se puede aplicar a la construcción de algoritmos que determinen si el lenguaje aceptado por un autómata es vacío o no, y para ver si es o no finito. Para lo primero se tiene que comprobar si el autómata acepta alguna palabra de longitud menor o igual que n. Para lo segundo si acepta una palabra de longitud mayor o igual que n y menor que $2n$. Como en ambos

casos, el número de palabras que hay que comprobar es finito, estos algoritmos siempre paran. Sin embargo, en general será n ineficientes.

Existe otro procedimiento más eficiente para determinar si el lenguaje asociado a un autómata finito determinístico es o no finito. Lo vamos a explicar en función del diagrama de transición asociado. Consiste en eliminar previamente todos los estados que son inaccesibles desde el estado inicial (un estado es inaccesible desde el estado inicial si no existe un camino dirigido entre el estado inicial y dicho estado). Después de eliminar los estado inaccesibles se obtiene un autómata finito determinístico que acepta el mismo lenguaje que el inicial. Si en dicho autómata queda algún estado final, el lenguaje aceptado por el autómata no es vacío. Es vacío si no queda ningún estado final.

Para comprobar si es o no finito, se eliminan además los estados desde los que no se puede acceder a un estado final y las transiciones asociadas. Se obtiene así un autómata finito que puede ser no determinístico y que acepta el mismo lenguaje que el original. El lenguaje aceptado será infinito si y solo si existe un ciclo en el diagrama de transición del nuevo autómata.

Teorema 11 *Existe un algoritmo para determinar si dos autómatas finitos determinísticos aceptan el mismo lenguaje.*

Demostración.-

Sean M_1 y M_2 dos autómatas finitos determinísticos. Entonces de forma algorítmica se puede construir un autómata, M, que acepte el lenguaje

$$L(M) = (L(M_1) \cap \overline{L(M_2)}) \cup (\overline{L(M_1)} \cap L(M_2))$$

Entonces el comprobar si $L(M_1) = L(M_2)$ es equivalente a comprobar si $L(M) \neq \emptyset$, lo cual se puede hacer también de forma algorítmica. ■

3.4. Teorema de Myhill-Nerode. Minimización de Autómatas

Sea $L \subseteq A^*$ un lenguaje arbitrario. Asociado a este lenguaje L podemos definir una relación de equivalencia R_L en el conjunto A, de la siguiente forma:

Si $x, y \in A^*$, entonces (xR_Ly) si y solo si $(\forall z \in A^*, (xz \in L \Leftrightarrow yz \in L))$

Esta relación de equivalencia dividirá el conjunto A^* en clases de equivalencia.

El número de clases de equivalencia se llama índice de la relación.

Ejemplo 47 *Sea* $L = 0^*10^*$*, entonces tenemos que*

$$00R_L000$$

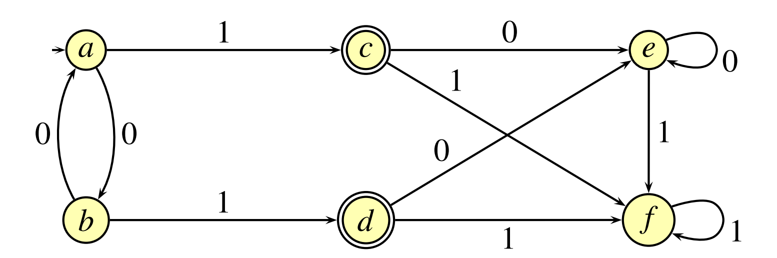


Figura 3.1: Autómata que acepta el lenguaje L

$$001 R_L 1$$

$$01 \not R_L 0$$

$$11 R_L 101$$

También se puede definir una relación de equivalencia, R_M, en A^* asociada a un autómata finito determinístico $M = (Q, A, \delta, q_0, F)$ cualquiera de la siguiente forma,

Si $u, v \in A^*$, entonces $u R_M v$ si y solo si $(\delta'(q_0, u) = \delta'(q_0, v))$

Esta relación de equivalencia divide también el lenguaje A en clases de equivalencia.

Ejemplo 48 *Consideremos el autómata de la Figura 3.1, que acepta el mismo lenguaje, L, del ejemplo anterior. Ahora tenemos que*

$$00 R_M 0000$$

$$0010 R_M 00010$$

$$00 R_M 0000$$

$$0010 R_M 00010$$

pero R_L *y* R_M *no son exactamente la misma relación de equivalencia.*

Por ejemplo, $000 R_L 00$ *pero* $000 \not R_M 00$.

En general, esto se va a verificar siempre. Si tenemos un lenguaje regular L y un autómata finito determinístico M que acepta este lenguaje entonces,

$$u R_M v \Rightarrow u R_L v$$

pero la implicación inversa no se va a verificar siempre. Solo, como veremos más adelante, para los autómatas minimales.

El índice (n. de clases de equivalencia) de R será a lo más el número de estados del autómata finito que sean accesibles desde el estado inicial. En efecto, si $q \in Q$, y este estado es accesible, entonces definirá una clase de equivalencia:

$$[q] = \{x \in A^* : \delta'(q_0, x) = q\}$$

Esta clase es no vacía ya que $\exists x \in A$ tal que $\delta'(q_0, x) = q$ (q es accesible desde q_0).

Definición 40 *Una relación de equivalencia R en A^* se dice que es invariante por la derecha para la concatenación si y solo si $((uRv) \Rightarrow (\forall z \in A^*, xzRyz))$.*

Proposición 6

- *Sea $L \subseteq A^*$, entonces R_L es invariante por la derecha.*
- *Si M es un autómata finito, entonces R_M es invariante por la derecha.*

El siguiente teorema es la base para la construcción de autómatas finitos minimales.

Teorema 12 (Teorema de Myhill-Nerode) *Si $L \subseteq A^*$ entonces las tres siguientes afirmaciones son equivalentes*

1. *L es aceptado por un autómata finito*
2. *L es la unión de algunas de las clases de equivalencia de una relación de equivalencia en A^* de índice finito que sea invariante por la derecha.*
3. *La relación de equivalencia R_L es de índice finito.*

Demostración.-
Demostraremos $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$.

$1 \Rightarrow 2$ Si M es un autómata finito determinístico que acepta el lenguaje L. Entonces la

relación de equivalencia R_M tiene un número finito de clases de equivalencia (tantos como estados accesibles desde el estado inicial tenga M) y además es invariante por la derecha.

Solo hay que probar que siendo $[x]_M$ la clase de equivalencia asociada a x ($[x]_M = \{y : xR_M y\}$ entonces

$$L = \bigcup_{\delta(q_0,x) \in F} [x]_M$$

$2 \Rightarrow 3$ Sea R la relación de equivalencia que verifica 2. Vamos a probar que

$$xRy \Rightarrow xR_Ly$$

En efecto, si xRy, entonces si $z \in A^*$, por ser R invariante por la derecha, tenemos que $xzRyz$. Es decir, xz, yz pertenecen a la misma clase de equivalencia. Como L es la unión de ciertas clases de equivalencia de la relación R, entonces todos los elementos de una misma clase pertenecen o no pertenecen al mismo tiempo al lenguaje L. De esto se deduce que $xz \in L \Leftrightarrow yz \in L$.

En conclusión, si xRy, entonces para todo $z \in A^*$, se tiene que $xz \in L \Leftrightarrow yz \in L$. Por tanto xR_Ly.

Una vez demostrado esto, veremos que R_L tiene memos clases de equivalencia que R. Más concretamente, veremos que si $[x]_R = [y]_R$ entonces $[x]_L = [y]_L$. Pero esto es inmediato, ya que si $[x]_R = [y]_R$, entonces xRy. Y acabamos de demostrar que de aquí se deduce que xR_Ly. Y por tanto, $[x]_L = [y]_L$.

Por último, como R es de índice finito y R_L tiene menos clases que R, se deduce que R_L es de índice finito.

$3 \Rightarrow 1$ Construiremos un autómata a partir de la relación R_L.

El autómata será $M = (Q, A, \delta, q_0, F)$, donde

- $Q = \{[x] : x \in A^*\}$
- $\delta([x]_L, a) = [xa]_L$
  Esta definición es consistente ya que, siendo R_L invariante por la derecha, si $[x]_L = [y]_L$ entonces $[xa]_L = [ya]_L$.
- $q_0 = [\varepsilon]_L$,
- $F = \{[x]_L : x \in L\}$.
  F está bien definido, ya que si $[x]_L = [y]_L$, entonces xR_Ly, y por tanto, $x\varepsilon \in L \Leftrightarrow y\varepsilon \in L$, es decir, $x \in L \Leftrightarrow y \in L$.

Este autómata acepta el lenguaje L. En efecto, $\delta(q_0, x) = [x]_L$ y $[x]_L \in F \Leftrightarrow x \in L$. Por tanto una palabra $x \in A^*$ es aceptada cuando $x \in L$.

■

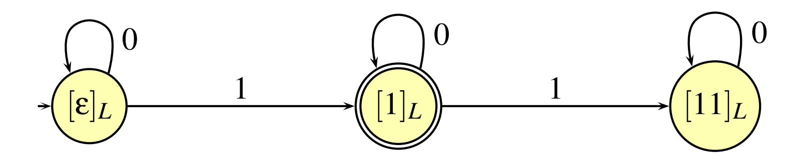


Figura 3.2: Autómata asociado a la relación R_L

Ejemplo 49 *Si consideramos el lenguaje L dado por la expresión regular* 0^*10^**, entonces la relación* R_L *divide al conjunto de las cadenas en tres clases de equivalencia:*

$$[\varepsilon]_L = C_1, \quad [1]_L = C_2, \quad [11]_L = C_3$$

Si consideramos el autómata asociado a este lenguaje (ver Fig. 3.1), este autómata divide A^* *en seis clases de equivalencia.*

$C_a = (00)^*$ $\quad C_b = 0(00)^*$ $\quad C_c = (00)^*1$

$C_d = (00)^*01$ $\quad C_e = 0^*100^*$ $\quad C_f = 0^*10^*1(0+1)^*$

$L = C_c \cup C_d \cup C_e$*, la unión de las tres clases correspondientes a los estados finales.*

De acuerdo con el teorema anterior, con la relación de equivalencia R_L podemos construír un autómata finito que acepta el mismo lenguaje L. Este autómata viene dado por el diagrama de transición de la figura 3.2,

3.4.1. Minimización de Autómatas

Definición 41 *Un autómata finito determinístico se dice minimal si no hay otro autómata finito determinístico que acepte el mismo lenguaje y tenga menos estados que el.*

En el ejemplo que vimos despues del teorema de Myhill-Nerode, a partir de la relación R_L asociada a un lenguaje L, construímos un autómata que aceptaba ese lenguaje y que tenía *pocos estados*. En particular tenía menos estados que el autómata original. El siguiente teorema demuestra que es precisamente un autómata minimal.

Teorema 13 *Si L es un conjunto regular y* R_L *la relación de equivalencia asociada, entonces el autómata construido en el teorema anterior es minimal y único salvo isomorfismos.*

Demostración.-

Comprobaremos unicamente que es minimal.

En efecto, si M' es el autómata construído según el teorema anterior y M un autómata cualquiera que acepta el lenguaje L entonces tenemos que se verifica

$$xR_My \Rightarrow xRy$$

Por tanto R_L tiene menos clases que R_M.
El teorema queda demostrado si tenemos en cuenta que:
N. de estados de $M \geq$ N. de clases de $R_M \geq$ N. de clases de $R_L =$ N. de estados de M'

■

El teorema anterior nos permite encontrar el autómata minimal asociado a un lenguaje, partiendo de la relación de equivalencia de dicho lenguaje. El problema fundamental es que normalmente no tenemos dicha relación y normalmente no es fácil de calcular. Lo que solemos tener es un autómata finito determinístico que acepta el lenguaje L y lo que nos interesa es transformarlo en un autómata minimal. Esto se puede hacer de forma algorítmica. El procedimiento de minimización está basado en el hecho de que los estados del autómata minimal construído a partir de la relación R_L están formados por uniones de estados de un autómata M cualquiera que acepte el mismo lenguaje. Los estados que se pueden identificar y unir en un solo estado son los llamado estados indistinguibles, que se definen a continuación.

Definición 42 *Si $M = (Q, A, \delta, q_0, F)$ es un autómata finito determinístico y q_i, q_j son dos estados de Q, se dice que q_i y q_j son indistinguibles si y solo si $\forall u \in A^*, \delta'(q_i, u) \in F \Leftrightarrow \delta'(q_j, u) \in F$.*

Es decir, es indiferente estar en dos estados indistinguibles para el único objetivo que nos interesa: saber si vamos a llegar a un estado final o no.

El siguiente algoritmo identifica las parejas de estados indistinguibles. Supone que no hay estados inaccesibles.

Para el algoritmo, asociaremos a cada pareja de estados accesibles del autómata una variable booleana: *marcado*, y una lista de parejas. Al principio todas las variables booleanas están a falso y las listas vacías. Los pasos del algoritmo son como siguen,

1. Eliminar estados inaccesibles.
2. Para cada pareja de estados accesibles $\{q_i, q_j\}$
   3. Si uno de ellos es final y el otro no, hacer la variable booleana
      asociada igual a true.
4. Para cada pareja de estados accesibles $\{q_i, q_j\}$
   5. Para cada símbolo a del alfabeto de entrada
      6. Calcular los estados q_k y q_l a los que evoluciona el
         autómata desde q_i y q_j leyendo a
      7. Si $q_k \neq q_l$ entonces
         8. Si la pareja $\{q_k, q_l\}$ está marcada entonces se
            marca la pareja $\{q_i, q_j\}$ y recursivamente se marcan también todas
            las parejas en la lista asociada.
         9. Si la pareja $\{q_k, q_l\}$ no está marcada, se añade la pareja

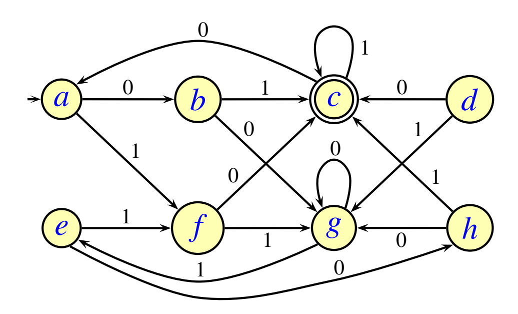


Figura 3.3: Autómata para minimizar

$\{q_i, q_j\}$ a la lista asociada a la pareja $\{q_k, q_l\}$.

Al final del algoritmo, todas las parejas de estados marcados son distinguibles y los no marcados indistinguibles.

Una vez identificados los estados indistinguibles, el autómata minimal se puede construir identificando los estados indistinguibles. Más concretamente, si el autómata original es $M = (Q, A, \delta, q_0, F)$, R es la relación de equivalencia de indistinguibilidad entre estados y $[q]$ la clase de equivalencia asociada al estado q, entonces el nuevo autómata, $M_m = (Q_m, A, \delta_m, q_0^m, F_m)$ tiene los siguientes elementos,

- $Q_m = \{[q] : q$ es accesible desde $q_0\}$
- $F_m = \{[q] : q \in F\}$
- $\delta_m([q], a) = [\delta(q, a)]$
- $q_0^m = [q_0]$

Se puede demostrar el siguiente teorema.

Teorema 14 *Si M es un autómata finito determinístico sin estados inaccesibles, el autómata M_m construido anteriormente es minimal.*

Ejemplo 50 *Minimizar el autómata de la Figura 3.3.*

Primero eliminamos el estado inaccesible d y obtenemos el autómata de la figura **??**. *A continuación organizamos los datos del algoritmo en una tabla triangular como la de la Figura 3.5. En las casillas vamos anotando la lista asociada a cada pareja y si están o no marcadas.*

El resultado se puede ver en la tabla 3.6. Esta nos indica que los estados a y e son equivalentes, así como los estados b y h: $a \equiv e, b \equiv h$. Identificando estos estados, obtenemos el autómata de la figura 3.7.

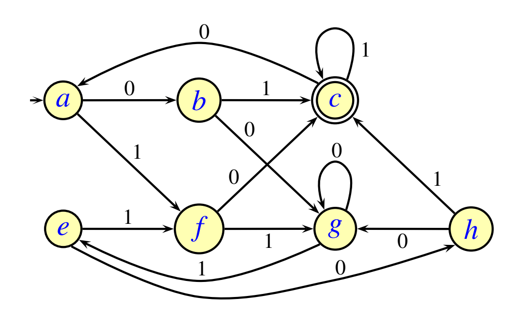


Figura 3.4: Autómata para minimizar, después de eliminar estados inaccesibles.

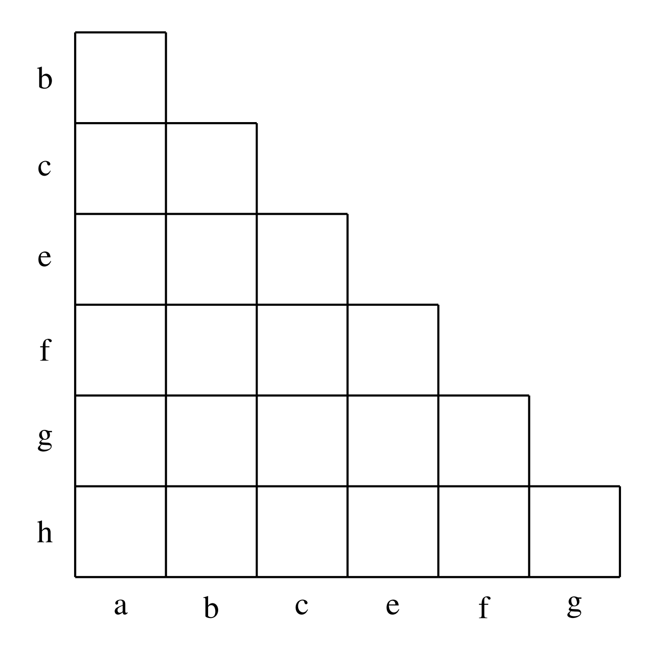


Figura 3.5: Tabla de minimización de autómata

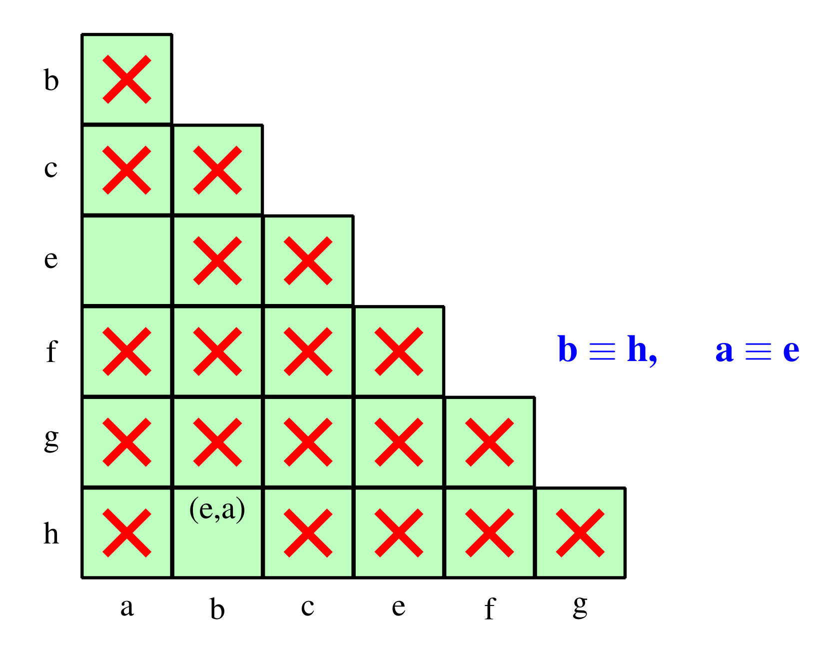


Figura 3.6: Estado final de la tabla de minimización de autómata

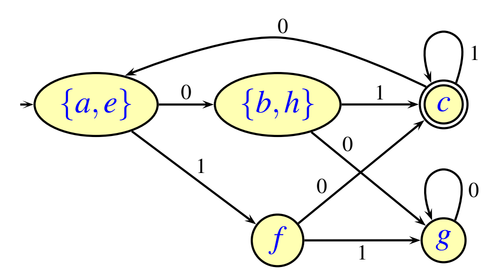


Figura 3.7: Autómata minimal

Capítulo 4

Gramáticas Libres de Contexto

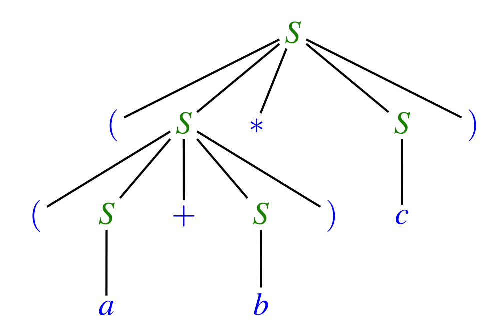


Figura 4.1: Arbol de Derivación

4.1. Arbol de Derivación y Ambigüedad

Consideremos una gramática $G = (V,T,P,S)$ donde $V = \{S\}, T = \{a,b,c,\emptyset,\varepsilon,(,),*,+\}$ y las producciones son

$S \rightarrow a|b|c|\emptyset|\varepsilon$
$S \rightarrow (S+S)$
$S \rightarrow (S*S)$

Esta es una gramática libre de contexto: En la parte izquierda de las producciones solo aparece una variable. Al lenguage generado por esta gramática pertenece la palabra $((a+b)*c)$. Solo hay que aplicar la siguiente cadena de producciones

$$S \Rightarrow (S*S) \Rightarrow ((S+S)*S) \Rightarrow ((a+S)*S) \Rightarrow ((a+b)*S) \Rightarrow ((a+b)*c)$$

Una palabra nos puede ayudar a determinar si una palabra pertenece a un determinado lenguaje, pero también a algo más: a determinar la estructura sintáctica de la misma. Esta viene dada por lo que llamaremos árbol de derivación. Este se construye a partir de la cadena de derivaciones de la siguiente forma. Cada nodo del árbol va a contener un símbolo. En el nodo raíz se pone el símbolo inicial S.

Entonces, si a este nodo se le aplica una determinada regla $S \rightarrow \alpha$, entonces para cada símbolo que aparezca en α se añade un hijo con el símbolo correspondiente, situados en el orden de izquierda a derecha. Este proceso se repite para todo paso de la derivación. Si la derivación se aplica a una variable que aparece en un nodo, entonces se le añaden tantos hijos como símbolos tenga la parte derecha de la producción. Si la parte derecha es una cadena vacía, entonces se añade un solo hijo, etiquetado con ε. En cada momento, leyendo los nodos de izquierda a derecha se lee la palabra generada.

En nuestro caso tenemos el árbol de derivación de la Figura 4.1.

Un árbol se dice completo cuando todas las etiquetas de los nodos hojas son símbolos terminales o bien la cadena vacía.

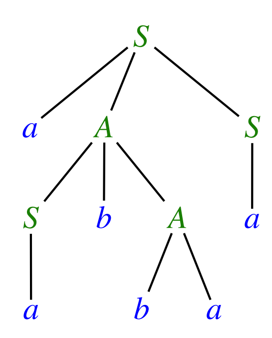


Figura 4.2: Arbol de Derivación de *aabbaa*

Ejemplo 51 *Consideremos la gramática*

$$S \to aAS, \quad S \to a, \quad A \to SbA, \quad A \to SS, \quad A \to ba$$

y la palabra aabbaa. Esta palabra tiene una derivación que tiene asociado el árbol de la Figura 4.2

Aunque toda cadena de derivaciones lleva asociado un solo árbol, éste puede provenir de varias cadenas de derivaciones distintas. Sin embargo, siempre se pueden distinguir las siguientes:

- *Derivación por la izquierda.-* Cuando en cada paso siempre se sustituye primero la primera variable (más a la izquierda) de la palabra que tenemos.
- *Derivación por la derecha.-* Cuando en cada paso siempre se sustituye primero la última variable (más a la derecha) de la palabra que tenemos.

Ejemplo 52 *Al árbol de derivación de la palabra aabbaa, le corresponden las siguientes derivaciones:*

- *Derivación por la izquierda:*

$$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS \Rightarrow aabbaa$$

- *Derivación por la derecha:*

$$S \Rightarrow aAS \Rightarrow aAa \Rightarrow aSbAa \Rightarrow aSbbaa \Rightarrow aabbaa$$

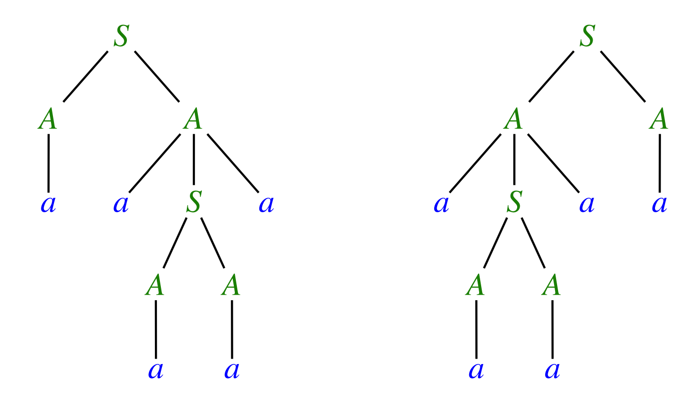


Figura 4.3: Dos árboles de derivación para a^5

La existencia en una gramática de varias derivaciones para una misma palabra no produce ningún problema, siempre que den lugar al mismo árbol. Sin embargo, existe otro problema que si puede ser más grave: la ambigüedad. Una gramática se dice ambigua si existen dos árboles de derivación distintos para una misma palabra. Esto es un problema, ya que la gramática no determina la estructura sintáctica de los elementos de la palabra, no determina como se agrupan los distintos elementos para formar la palabra completa.

Ejemplo 53 *La gramática,*

$$S \to AA, \quad A \to aSa, \quad A \to a$$

es ambigua: la palabra a^5 tiene dos árboles de derivación distintos (ver Fig. 4.3).

Es suficiente que haya una palabra con dos árboles de derivación distintos para que la gramática sea ambigua.

El lenguaje generado por esta gramática no es inherentemente ambiguo. Esto quiere decir que existe otra gramática de tipo 2 no ambigua y que genera el mismo lenguaje. El lenguaje generado es $\{a^{2+3i} : i \geq 0\}$ y otra gramática no ambigua que también geneta este lenguaje es

$$S \to aa, \quad S \to aaU, \quad U \to aaaU, \quad U \to aaa$$

Aquí a solo tiene un árbol de derivación asociado (ver Fig. 4.4).

Ejemplo 54 *La gramática*

$$E \to I, \ \ E \to I - E, \ \ E \to E - I, \ \ I \to a|b|c|d$$

es ambigua. La palabra $a - b - c$ admite dos árboles de derivación (ver Fig. 4.5).

Se puede eliminar la ambigüedad eliminando la producción $E \to I - E$.

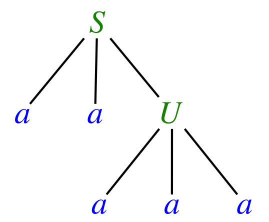


Figura 4.4: Arbol de derivación para a^5

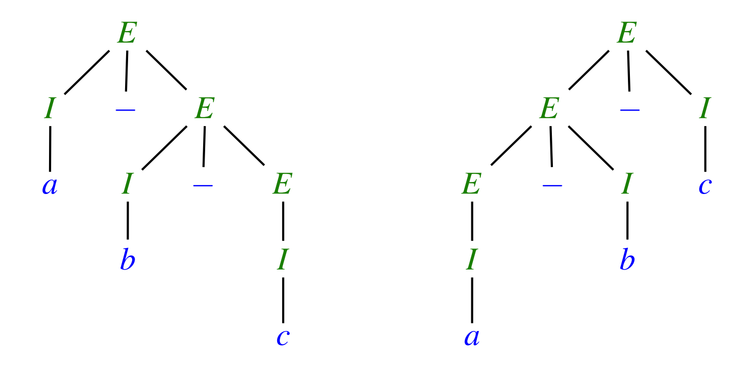


Figura 4.5: Dos árboles de derivación para $a-b-c$

Existen lenguajes inherentemente ambiguos, como es el siguiente

$$L = \{a^n b^n c^m d^m : n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n : n \geq 1, m \geq 1\}$$

Este lenguaje es de tipo 2 o libre del contexto, ya que puede puede generarse por la gramática
$S \to AB$, $A \to ab$, $A \to aAb$,
$B \to cd$, $B \to cBd$,
$S \to aCd$, $C \to aCd$, $C \to bDc$,
$C \to bc$, $D \to bDc$, $D \to bc$
Entonces la palabra $aabbccdd$ puede tener dos derivaciones (ver Fig. 4.6).

Aunque no lo demostraremos aquí esta ambigüedad no se puede eliminar. Pueden existir otras gramáticas de tipo 2 que generen el mismo lenguaje, pero todas será n ambiguas.

4.2. Simplificación De Las Gramáticas Libres De Contexto

Para un mismo lenguage de tipo 2 existen muchas gramáticas libres de contexto que lo generan. Estas será n de formas muy diversas por lo que, en general se hace muy difícil trabajar

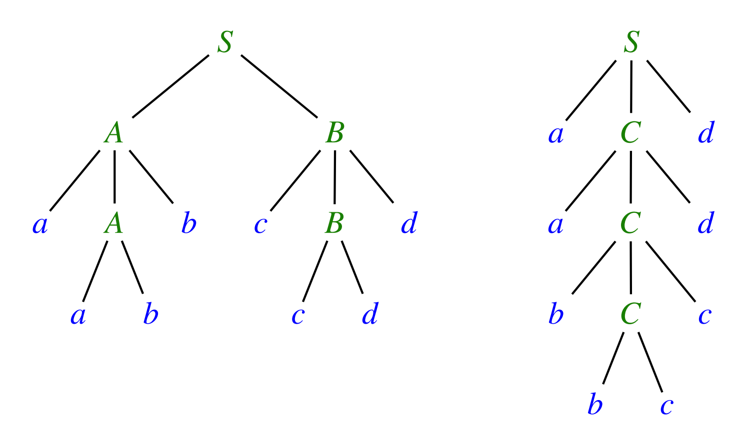


Figura 4.6: Dos árboles de derivación para *aabbccdd*

con ellas. Por este motivo interesa simplificarlas lo mas posible y definir unas formas normales para las gramáticas que las hagan mas homogéneas.

4.2.1. Eliminación de Símbolos y Producciones Inútiles

Un símbolo $X \in (V \cup T)$ se dice útil si y solo si existe una cadena de derivaciones en G tal que

$$S \Rightarrow \alpha X \beta \Rightarrow w \in T$$

es decir si interviene en la derivación de alguna palabra del lenguaje generado por la gramática. Una producción se dice útil si y solo si todos sus símbolos son útiles. Esto es equivalente a que pueda usarse en la derivación de alguna palabra del lenguaje asociado a la gramática. Esta claro que eliminando todos los símbolos y producciones inútiles el lenguaje generado por la gramática no cambia.

El algoritmo para eliminar los símbolos y producciones inútiles consta de dos pasos fundamentales:

1. Eliminar las variables desde las que no se puede llegar a una palabra de T y las producciones en las que aparezcan.

2. Eliminar aquellos símbolos que no sean alcanzables desde el estado inicial, S, y las producciones en las que estos aparezcan.

El primer paso se realiza con el siguiente algoritmo (V’ es un conjunto de variables):

1. $V' = \emptyset$
2. Para cada producción de la forma $A \to w$, A se introduce en V'.
3. Mientras V' cambie
   4. Para cada producción $B \to \alpha$
      5. Si todas las variables de α pertenecen a V',
         B se introduce en V'
6. Eliminar las variables que esten en V y no en V'
7. Eliminar todas las producciones donde aparezca una
   variable de las eliminadas en el paso anterior

El segundo paso se realiza con el siguiente algoritmo:

- V'' y J son conjuntos de variables. J son las variables por analizar.
- T' es un conjunto de símbolos terminales

1. $J = \{S\}$
   $V'' = \{S\}$
   $T' = \emptyset$
2. Mientras $J \neq \emptyset$
   3. Extraer un elemento de $J : A$, $(J = J - \{A\})$.
   4. Para cada produccion de la forma $A \to \alpha$
      5.Para cada variable B en α
         6. Si B no est en V'' añadir B a J y a V''
      7. Poner todos los simbolos terminales de α en T'
8. Eliminar todas las variables que no esten en V''
   y todos los simbolos terminales que no esten en T'.
9. Eliminar todas las producciones donde aparezca
   un simbolo o variable de los eliminado

Ejemplo 55 *Es importante aplicar los algoritmos anteriores en el orden especificado para que se garantice que se eliminan todos los símbolos y variables inútiles. Como ejemplo de lo anterior, supongamos que tenemos la gramática dada por*

$$S \to AB, \quad S \to a, \quad A \to a$$

En el primer algoritmo se elimina B y la producción $S \to AB$.

Entonces en el segundo se elimina la variable A y la producción $A \to a$.

Sin embargo, si aplicamos primero el segundo algoritmo, entonces no se elimina nada. Al aplicar despues el primero de los algoritmos se elimina B y la producción $S \to AB$. En definitiva, nos queda la gramática

$$S \to a, A \to a$$

donde todavía nos queda la variable inútil A.

Ejemplo 56 *Eliminar símbolos y producciones inútiles de la gramática*

$$S \to gAe, \quad S \to aYB, \quad S \to cY,$$
$$A \to bBY, \quad A \to ooC, \quad B \to dd,$$
$$B \to D, \quad C \to jVB, \quad C \to gi,$$
$$D \to n, \quad U \to kW, \quad V \to baXXX,$$
$$V \to oV, \quad W \to c, \quad X \to fV, \quad Y \to Yhm$$

Primero se aplica el algoritmo primero.

Inicialmente V' tiene las variables $V' = \{B, D, C, W\}$.

En la siguiente iteración añadimos a V' las variables que se alcanzan desde estas: A y W. V' queda igual a $\{B, D, C, W, A, U\}$.

En la siguiente $V' = \{B, D, C, W, A, U, S\}$.

En la siguiente $V' = \{B, D, C, W, A, U, S\}$.

Como V' no cambia ya se ha terminado el ciclo. Estas son las variables desde las que se alcanza una cadena de símbolos terminales. El restos de las variables: X, Y, V, son inútiles y se pueden eliminar. También se eliminan las producciones asociadas. La gramática resultante es:

$$S \to gAe, \quad A \to ooC,$$
$$B \to dd, \quad B \to D, \quad C \to gi,$$
$$D \to n, \quad U \to kW, \quad W \to c$$

A esta gramática le aplicamos el segundo algoritmo

$J = \{S\}, V'' = \{S\}, T' = \emptyset$

Tomando S de J y ejecutando las instrucciones del ciclo principal nos queda

$J = \{A\}, V'' = \{S, A\}, T' = \{g, e\}$

Tomando A de J, tenemos

$J = \{C\}, V'' = \{S, A, C\}, T' = \{g, e, o\}$

Sacando C de J, obtenemos

$J = \emptyset, V'' = \{S, A, C\}, T' = \{g, e, o, i\}$

Como $J = \emptyset$ se acaba el ciclo. Solo nos queda eliminar las variables inútiles: B, D, U, W, los símbolos terminales inútiles: n, k, c, d, y las producciones donde estos aparecen. La gramática queda

$$S \to gAe, \quad A \to ooC, \quad C \to gi$$

En esta gramática solo se puede generar una palabra: googige.

Si el lenguaje generado por una gramática es vacío, esto se detecta en que la variable S resulta inútil en el primer algoritmo. En ese caso se pueden eliminar directamente todas las producciones, pero no el símbolo S.

Ejemplo 57 *eliminar símbolos y producciones inútiles de la gramática*

$$S \to aSb, S \to ab, S \to bcD,$$

$$S \to cSE, E \to aDb, F \to abc, E \to abF$$

4.2.2. Producciones Nulas

Las producciones nulas son las de la forma $A \to \varepsilon$. Vamos a tratar de eliminarlas sin que cambie el lenguaje generado por la gramática ni la estructura de los árboles de derivación. Evidentemente si $\varepsilon \in L(G)$ no vamos a poder eliminarlas todas sin que la palabra nula deje de generarse. Así vamos a dar un algoritmo que dada una gramática G, construye una gramática G' equivalente a la anterior sin producciones nulas y tal que $L(G') = L(G) - \{\varepsilon\}$. Es decir, si la palabra nula era generada en la gramática original entonces no puede generarse en la nueva gramática. Primero se calcula el conjunto de las variables anulables:

H es el conjunto de las variables anulables

1. $H = \emptyset$
2. Para cada produccion $A \to \varepsilon$, se hace $H = H \cup \{A\}$
3. Mientras H cambie
   4. Para cada produccion $B \to A_1A_2\ldots A_n$,
      donde $A_i \in H$ para todo $i = 1,\ldots,n$, se hace $H = H \cup \{B\}$

Una vez calculado el conjunto, H, de las variables anulables, se hace lo siguiente:

1. Se eliminan todas las producciones nulas de la gramatica
2. Para cada produccion de la gramatica de la forma
   $A \to \alpha_1 \ldots \alpha_n$, donde $\alpha_i \in V \cup T$.
   3. Se elimina la produccion $A \to \alpha_1 \ldots \alpha_n$
   4. Se añaden todas las producciones de la forma $A \to \beta_1 \ldots \beta_n$

donde $\beta_i = \alpha_i$ si $\alpha_i \notin H$
$(\beta_i = \alpha_i) \vee (\beta_i = \varepsilon)$ si $\alpha \in H$
y no todos los β_i puedan ser nulos al mismo tiempo

G' es la gramática resultante después de aplicar estos dos algoritmos.

Si G generaba inicialmente la palabra nula, entonces, a partir de G', podemos construir una gramática G'' con una sola producción nula y que genera el mismo lenguaje que G. para ello se añade una nueva variable, S', que pasa a ser el símbolo inicial de la nueva gramática, G''. También se añaden dos producciones:

$$S' \rightarrow S, \quad S' \rightarrow \varepsilon$$

Ejemplo 58 *Eliminar las producciones nulas de la siguiente gramática:*

$$S \rightarrow ABb, S \rightarrow ABC, C \rightarrow abC,$$

$$B \rightarrow bB, B \rightarrow \varepsilon, A \rightarrow aA,$$

$$A \rightarrow \varepsilon, C \rightarrow AB$$

Las variables anulables despues de ejecutar el paso 2 del primer algoritmo son B y A. Al ejecutar el paso 3, resulta que C y S son también anulables, es decir $H = \{A, B, C, S\}$. Al ser S anulable la palabra vacía puede generarse mediante esta gramática. En la gramática que se construye con el segundo algoritmo esta palabra ya no se podrá generar. Al ejecutar los pasos 3 y 4 del segundo algoritmo para las distintas producciones no nulas de la gramática resulta

3. Se elimina la producción $S \rightarrow ABb$
4. Se añade $S \rightarrow ABb, S \rightarrow Ab, S \rightarrow Bb$

3. Se elimina $S \rightarrow ABC$
4. Se añade $S \rightarrow ABC, S \rightarrow AB, S \rightarrow AC,$
$S \rightarrow BC, S \rightarrow A, S \rightarrow B, S \rightarrow C$

3. Se elimina $C \rightarrow abC$
4. Se añade $C \rightarrow abC, C \rightarrow ab$

3. Se elimina $B \rightarrow bB$
4. Se añade $B \rightarrow bB, B \rightarrow b$

3. Se elimina $A \rightarrow aA$
4. Se añade $A \rightarrow aA, A \rightarrow a$

3. Se elimina $C \rightarrow AB$

4. Se añade $C \rightarrow AB, C \rightarrow A, C \rightarrow B$
En definitiva, la gramática resultante tiene las siguientes producciones:

$$S \rightarrow ABb, \quad S \rightarrow Ab, \quad S \rightarrow Bb, \quad S \rightarrow ABC, \quad S \rightarrow AB,$$
$$S \rightarrow AC, \quad S \rightarrow BC, \quad S \rightarrow A, \quad S \rightarrow B, \quad S \rightarrow C,$$
$$C \rightarrow abC, \quad C \rightarrow ab, \quad B \rightarrow bB, \quad B \rightarrow b, \quad A \rightarrow aA,$$
$$A \rightarrow a, \quad C \rightarrow AB, \quad C \rightarrow A, \quad C \rightarrow B$$

Ejemplo 59 *Sea la gramática*

$$S \rightarrow aHb, \quad H \rightarrow aHb, \quad H \rightarrow \varepsilon$$

La única variable anulable es H. Y la gramática equivalente, que resulta de aplicar el algoritmo 2 es:

$$S \rightarrow aHb, \quad H \rightarrow aHb, \quad S \rightarrow ab, \quad H \rightarrow ab$$

4.2.3. Producciones Unitarias

Las producciones unitarias son las que tienen la forma

$$A \rightarrow B$$

donde $A, B \in V$.

Veamos como se puede transformar una gramática G, en la que $\varepsilon \notin L(G)$, en otra gramática equivalente en la que no existen producciones unitarias.

Para ello, hay que partir de una gramática sin producciones nulas.

Entonces, se calcula el conjunto H de parejas (A, B) tales que B se puede derivar a partir de A: $A \Rightarrow B$. Eso se hace con el siguiente algoritmo

1. $H = \emptyset$
2. Para toda produccion de la forma $A \rightarrow B$,
   la pareja (A, B) se introduce en H.
3. Mientras H cambie
   4. Para cada dos parejas $(A, B), (B, C)$
      5. Si la pareja (A, C) no est en H
         (A, C) se introduce en H
6. Se eliminan las producciones unitarias
7. Para cada produccion $A \rightarrow \alpha$

8. Para cada pareja $(B,A) \in H$
   9. Se añade una produccion $B \rightarrow \alpha$

Ejemplo 60 *Consideremos la gramática resultante en el ejemplo 58 del apartado anterior*

$$S \rightarrow ABb, \quad S \rightarrow Ab, \quad S \rightarrow Bb, \quad S \rightarrow ABC, \quad S \rightarrow AB,$$

$$S \rightarrow AC, \quad S \rightarrow BC, \quad S \rightarrow A, \quad S \rightarrow B, \quad S \rightarrow C,$$

$$C \rightarrow abC, \quad C \rightarrow ab, \quad B \rightarrow bB, \quad B \rightarrow b, \quad A \rightarrow aA,$$

$$A \rightarrow a, \quad C \rightarrow AB, \quad C \rightarrow A, \quad C \rightarrow B$$

El conjunto H est formado por las parejas $\{(S,A),(S,B),(S,C),(C,A),(C,B)\}$.
Entonces se eliminan las producciones

$$S \rightarrow A, \quad S \rightarrow B, \quad S \rightarrow C, \quad C \rightarrow A, \quad C \rightarrow B$$

Se añaden a continuacion las producciones (no se consideran las repetidas)

$$S \rightarrow a, \quad S \rightarrow aA, \quad S \rightarrow bB, \quad S \rightarrow b, \quad S \rightarrow abC,$$

$$S \rightarrow ab, \quad C \rightarrow aA, \quad C \rightarrow a, \quad C \rightarrow bB, \quad C \rightarrow b$$

Ejemplo 61 *El lenguaje* $L = \{a^n b^n : n \geq 1\} \cup \{a^n b a^n : n \geq 1\}$ *viene generado por la siguiente gramática de tipo 2:*

$$S \rightarrow A, \quad S \rightarrow B, \quad A \rightarrow ab, \quad A \rightarrow aHb$$

$$H \rightarrow ab, \quad H \rightarrow aHb, \quad B \rightarrow aBa, B \rightarrow b$$

Las parejas resultantes en el conjunto H son $\{(S,A),(S,B)\}$. *La gramática sin producciones unitarias equivalente es:*

$$A \rightarrow ab, \quad A \rightarrow aHb, \quad H \rightarrow ab, \quad H \rightarrow aHb, \quad B \rightarrow aBa,$$

$$B \rightarrow b, \quad S \rightarrow ab, \quad S \rightarrow aHb, \quad S \rightarrow aBa, \quad S \rightarrow b.$$

4.3. Formas Normales

4.3.1. Forma Normal de Chomsky

Una gramática de tipo 2 se dice que est en forma normal de Chomsky si y solo si todas las producciones tienen la forma

$$A \to BC, \quad A \to a,$$

donde $A, B, C \in V$, $a \in T$.

Toda gram ática de tipo 2 que no acepte la palabra vacía se puede poner en forma normal de Chomsky. Para ello lo primero que hay que hacer es suprimir las producciones nulas y unitarias. Acontinuación se puede ejecutar el siguiente algoritmo:

1. Para cada produccion de la forma $A \to \alpha_1 \ldots \alpha_n, \quad \alpha_i \in (V \cup T), n \geq 2$
   2. Para cada α_i , si α_i es terminal: $\alpha_i = a \in T$
      3. Se añade la produccion $C_a \to a$
      4. Se cambia α_i por C_a en $A \to \alpha_1 \ldots \alpha_n$
5. Para cada produccion de la forma $A \to B_1, \ldots B_m, \quad m \geq 3$
   6. Se añaden $(m-2)$ variables $D_1, D_2, \ldots, D_{m-2}$ (distintas para cada produccion)
   7. La produccion $A \to B_1 \ldots B_m$ se reemplaza por
      $A \to B_1 D_1, \quad D_1 \to B_2 D_2, \quad \ldots, \quad D_{m-2} \to B_{m-1} B_m$

Ejemplo 62 *Sea la Gramática $G = (\{S, A, B\}, \{a, b\}, P, S)$ dada por las producciones*

$$S \to bA \mid aB, \quad A \to bAA \mid AS \mid a, \quad B \to aBB \mid bS \mid b$$

Para pasarla a forma normal de Chomsky, en el ciclo asociado al paso 1 se añaden las producciones

$$C_a \to a, \quad C_b \to b$$

y las anteriores se transforman en

$$S \to C_b A \mid C_a B, \quad A \to C_b AA \mid AS \mid C_a, \quad B \to C_a BB \mid C_b S \mid C_b b$$

Al aplicar el paso asociado al paso 5, la gramática queda

$$S \to C_b A \mid C_b B, \quad A \to C_b D_1 \mid AS \mid a, \quad D_1 \to AA,$$
$$B \to C_a E_1 \mid C_b S \mid b, \quad E_1 \to B, \quad C_a \to a, \quad C_b \to b$$

Con esto la gramática ya está en forma normal de Chomsky.

4.3.2. Forma Normal de Greibach

Una gramática se dice que est en forma normal de Greibach si y solo si todas las producciones tienen la forma

$$A \rightarrow a\alpha$$

donde $a \in T$, $\alpha \in V^*$. Toda gramática de tipo 2 que no acepte la palabra vacía se puede poner en forma normal de Greibach. Para ello hay que partir de una gramática en forma normal de Chomsky y aplicarle el siguiente algoritmo. En realidad no es necesario que la gramática esté en forma normal de Chomsky. Basta que todas las producciones sean de uno de los tipos siguientes:

- $A \rightarrow a\alpha, \quad a \in T, \alpha \in V^*$.
- $A \rightarrow \alpha, \quad \alpha \in V^*$.

Claro está, en una gramática en forma normal de Chomsky, todas las producciones son de alguno de estos dos tipos.

En este algoritmo se supone que el conjunto de variables de la gramática está numerado: $V = \{A_1, \ldots, A_m\}$.

El algoritmo se basa en dos operaciones básicas. La primera es eliminar una producción, $A \rightarrow B\alpha$ de la gramática G, donde $A \neq B$. Esto se hace con los siguientes pasos:

1. Eliminar $A \rightarrow B\alpha$
2. Para cada produccion $B \rightarrow \beta$
   3. Añadir $A \rightarrow \beta\alpha$

La idea básica de esta operación es la siguiente. Si aplicamos $A \rightarrow B\alpha$, entonces, después tenemos que realizar una sustitución de B, mediante una producción $B \rightarrow \beta$. Uniendo las dos sustituciones, obtenemos $A \Rightarrow B\alpha \Rightarrow \beta\alpha$. Es decir, A se puede sustituir por $\beta\alpha$. Si eliminamos $A \rightarrow B\alpha$, esta sustitución ya no se puede hacer, por lo que si no queremos perder la posibilidad de eliminar ninguna palabra después de esta eliminación, tenemos que añadir producciones que nos permitan hacer esta sustitución. Por ese motivo añadimos $A \rightarrow \beta\alpha$, que permite realizar los dos pasos de antes en uno sólo.

La otra operación básica consiste en eliminar todas las producciones del tipo $A \rightarrow A\alpha$ donde $\alpha \in V^*$. Esto se hace siguiendo los siguiente pasos:

1. Añadir una nueva variable B_A
2. Para cada produccion $A \rightarrow A\alpha$
   3. Añadir $B_A \rightarrow \alpha$ y $B_A \rightarrow \alpha B_A$

4. Eliminar $A \rightarrow A\alpha$
5. Para cada produccion $A \rightarrow \beta$ β no empieza pro A
6. Añadir $A \rightarrow \beta B_A$

Llamemos ELIMINA$_1$($A \rightarrow B\alpha$) a la función que reliza el primer paso y ELIMINA$_2$(A) a la función que reliza el segundo paso. Si si llama a ELIMINA$_2$(A_j), la variable que añadimos la notaremos como B_j.

En estas condiciones vamos a realizar un algoritmo, al final del cual todas las producciones tengan una forma que corresponda a alguno de los patrones siguientes:

– $A \rightarrow a\alpha, \quad a \in T, \alpha \in V^*$.

– $A_i \rightarrow A_j\alpha, \quad j > i, \alpha \in V^*$.

– $B_j \rightarrow A_i\alpha, \quad \alpha \in V^*$

El algoritmo es como sigue:

1. Para cada $k = 1, \ldots, m$
2. Para cada $j = 1, \ldots, k-1$
3. Para cada produccion $A_k \rightarrow A_j\alpha$
4. ELIMINA$_1$($A_k \rightarrow A_j\alpha$)
5. Si existe alguna produccion de la forma $A_k \rightarrow A_k\alpha$
6. ELIMINA$_2$(A_k)

A continuación se puede eliminar definitivamente la recursividad por la izquierda con el siguiente algoritmo pasando a forma normal de Greibach

1. Para cada $i = m-1, \ldots, 1$
2. Para cada produccion de la forma $A_i \rightarrow A_j\alpha, \quad j > i$
3. ELIMINA$_1$($A_i \rightarrow A_j\alpha$)
4. Para cada $i = 1, 2, \ldots, m$
5. Para cada produccion de la forma $B_j \rightarrow A_i\alpha$.
6. ELIMINA$_1$($B_j \rightarrow A_i\alpha$)

El resultado del segundo algoritmo es ya una gramática en forma normal de Greibach.

Ejemplo 63 *Pasar a forma normal de Greibach la gramática dada por las producciones*

$$A_1 \rightarrow A_2A_3, \quad A_2 \rightarrow A_3A_1, \quad A_2 \rightarrow b, \quad A_3 \rightarrow A_1A_2, \quad A_3 \rightarrow a$$

Aplicamos ELIMINA$_1$ *a* $A_3 \rightarrow A_1A_2$.
Se elimina esta producción y se añade: $A_3 \rightarrow A_2A_3A_2$
Queda:

$$A_1 \rightarrow A_2A_3, \quad A_2 \rightarrow A_3A_1, \quad A_2 \rightarrow b,$$

$$A_3 \rightarrow a, \quad A_3 \rightarrow A_2A_3A_2$$

Aplicamos ELIMINA$_1$ *a* $A_3 \rightarrow A_2A_3A_2$
Se elimina esta producción y se añaden: $A_3 \rightarrow A_3A_1A_3A_2, \quad A_3 \rightarrow bA_3A_2$
Queda:

$$A_1 \rightarrow A_2A_3, \quad A_2 \rightarrow A_3A_1, \quad A_2 \rightarrow b,$$

$$A_3 \rightarrow a, \quad A_3 \rightarrow A_3A_1A_3A_2, \quad A_3 \rightarrow bA_3A_2$$

Aplicamos ELIMINA$_2$ *a* A_3
Se añade B_3 *y las producciones* $B_3 \rightarrow A_1A_3A_2, \quad B_3 \rightarrow A_1A_3A_2B_3$
Se elimina $A_3 \rightarrow A_3A_1A_3A_2$.
Se añaden las producciones: $A_3 \rightarrow aB_3, \quad A_3 \rightarrow bA_3A_2B_3$
Queda:

$$\begin{array}{llll} A_1 \rightarrow A_2A_3, & A_2 \rightarrow A_3A_1, & A_2 \rightarrow b, & A_3 \rightarrow a, \\ A_3 \rightarrow bA_3A_2 & B_3 \rightarrow A_1A_3A_2, & B_3 \rightarrow A_1A_3A_2B_3 & A_3 \rightarrow aB_3, \\ A_3 \rightarrow bA_3A_2B_3 & & & \end{array}$$

Se aplica ELIMINA$_1$ *a* $A_2 \rightarrow A_3A_1$.
Se elimina esta producción y se añaden:

$$A_2 \rightarrow aA_1, \quad A_2 \rightarrow aB_3A_1, \quad A_2 \rightarrow bA_3A_2B_3A_1, \quad A_2 \rightarrow bA_3A_2A_1$$

Queda:

$$\begin{array}{llll} A_1 \rightarrow A_2A_3, & A_2 \rightarrow b, & A_2 \rightarrow aA_1, & A_2 \rightarrow aB_3A_1, \\ A_2 \rightarrow bA_3A_2B_3A_1, & A_2 \rightarrow bA_3A_2A_1, & A_3 \rightarrow a, & A_3 \rightarrow bA_3A_2, \\ B_3 \rightarrow A_1A_3A_2, & B_3 \rightarrow A_1A_3A_2B_3 & A_3 \rightarrow aB_3, & A_3 \rightarrow bA_3A_2B_3 \end{array}$$

Se aplica ELIMINA$_1$ *a* $A_1 \rightarrow A_2A_3$.
Se elimina esta producción y se añaden:

$$A_1 \to bA_3, \quad A_1 \to aA_1A_3, \quad A_1 \to aB_3A_1A_3,$$
$$A_1 \to bA_3A_2B_3A_1A_3, \quad A_1 \to bA_3A_2A_1A_3$$

Queda:

$$\begin{array}{lll} A_2 \to b, & A_2 \to aA_1 & A_2 \to aB_3A_1, \\ A_2 \to bA_3A_2B_3A_1, & A_2 \to bA_3A_2A_1, & A_3 \to a, \\ A_3 \to bA_3A_2, & B_3 \to A_1A_3A_2, & B_3 \to A_1A_3A_2B_3, \\ A_3 \to aB_3, & A_3 \to bA_3A_2B_3, & A_1 \to bA_3, \\ A_1 \to aA_1A_3, & A_1 \to aB_3A_1A_3, & A_1 \to bA_3A_2B_3A_1A_3, \\ A_1 \to bA_3A_2A_1A_3 & & \end{array}$$

Se aplica ELIMINA$_1$ *a* $B_3 \to A_1A_3A_2$ *Se elimina esta producción y se añaden:*

$$\begin{array}{lll} B_3 \to bA_3A_3A_2, & B_3 \to aA_1A_3A_3A_2, & B_3 \to aB_3A_1A_3A_3A_2, \\ B_3 \to bA_3A_2B_3A_1A_3A_3A_2, & B_3 \to aB_3A_1A_3A_3A_2, & \end{array}$$

Queda:

$$\begin{array}{lll} A_2 \to b, & A_2 \to aA_1 & A_2 \to aB_3A_1, \\ A_2 \to bA_3A_2B_3A_1, & A_2 \to bA_3A_2A_1 & A_3 \to a, \\ A_3 \to bA_3A_2, & B_3 \to A_1A_3A_2B_3, & A_3 \to aB_3, \\ A_3 \to bA_3A_2B_3, & A_1 \to bA_3, & A_1 \to aA_1A_3, \\ A_1 \to aB_3A_1A_3, & A_1 \to bA_3A_2B_3A_1A_3, & A_1 \to bA_3A_2A_1A_3 \\ B_3 \to bA_3A_3A_2, & B_3 \to aA_1A_3A_3A_2, & B_3 \to aB_3A_1A_3A_3A_2, \\ B_3 \to bA_3A_2B_3A_1A_3A_3A_2, & B_3 \to aB_3A_1A_3A_3A_2, & \end{array}$$

Se aplica ELIMINA$_1$ *a* $B_3 \to A_1A_3A_2B_3$. *Se elimina esta producción y se añaden:*

$$\begin{array}{lll} B_3 \to bA_3A_3A_2B_3, & B_3 \to aA_1A_3A_3A_2B_3, & B_3 \to aB_3A_1A_3A_3A_2B_3, \\ B_3 \to bA_3A_2B_3A_1A_3A_3A_2B_3, & B_3 \to aB_3A_1A_3A_3A_2B_3, & \end{array}$$

Resultado:

$$\begin{array}{lll} A_2 \to b, & A_2 \to aA_1, & A_2 \to aB_3A_1, \\ A_2 \to bA_3A_2B_3A_1, & A_2 \to bA_3A_2A_1 & A_3 \to a, \\ A_3 \to bA_3A_2, & A_3 \to aB_3, & A_3 \to bA_3A_2B_3, \\ A_1 \to bA_3, & A_1 \to aA_1A_3, & A_1 \to aB_3A_1A_3, \\ A_1 \to bA_3A_2B_3A_1A_3, & A_1 \to bA_3A_2A_1A_3, & B_3 \to bA_3A_3A_2, \\ B_3 \to aA_1A_3A_3A_2, & B_3 \to aB_3A_1A_3A_3A_2, & B_3 \to bA_3A_2B_3A_1A_3A_3A_2, \\ B_3 \to aB_3A_1A_3A_3A_2, & B_3 \to bA_3A_3A_2B_3, & B_3 \to aA_1A_3A_3A_2B_3, \\ B_3 \to aB_3A_1A_3A_3A_2B_3, & B_3 \to bA_3A_2B_3A_1A_3A_3A_2B_3, & B_3 \to aB_3A_1A_3A_3A_2B_3 \end{array}$$

Capítulo 5

Autómatas con Pila

5.1. Definición de Autómata con Pila

Los lenguajes generados por las gramáticas libres de contexto también tienen un autómata asociado que es capaz de reconocerlos. Estos autómatas son parecidos a los autómatas finitos determinísticos, solo que ahora tendrán un dispositivo de memoria de capacidad ilimitada: una pila. A contincuación daremos la definición formal de autómata con pila no determinístico (APND). Al contrario que en los autómatas finitos, los autómatas con pila no detrminísticos y determinísticos no aceptan las mismas familias de lenguajes. Precisamente son los no determísticos los asociados con los lenguajes libres de contexto. Los determinísticos aceptan una familia mas restringida de lenguajes.

Definición 43 *Un autómata con pila no determinístico (APND) es una septupla* $(Q, A, B, \delta, q_0, Z_0, F)$ *en la que*

- Q *es un conjunto finito de estados*
- A *es un alfabeto de entrada*
- B *es un alfabeto para la pila*
- δ *es la función de transición*

$$\delta : Q \times (A \cup \{\varepsilon\}) \times B \longrightarrow \wp(Q \times B^*)$$

- q_0 *es el estado inicial*
- Z_0 *es el símbolo inicial de la pila*
- F *es el conjunto de estados finales*

La función de transición aplica cada estado, cada símbolo de entrada (incluyendo la cadena vacía) y cada símbolo tope de la pila en un conjunto de posibles movimientos. Cada movimiento parte de un estado, un símbolo de la cinta de entrada y un símbolo tope de la pila. El movimento en sí consiste en un cambio de estado, en la lectura del símbolo de entrada y en la subtitución del símbolo tope de la pila por una cadena de símbolos.

Ejemplo 64 *Sea el autómata* $M = (\{q_1, q_2\}, \{0, 1, c\}, \{R, B, G\}, \delta, q_1, R, \emptyset)$ *donde*

$$\begin{array}{ll}
\delta(q_1, 0, R) = \{(q_1, BR)\} & \delta(q_1, 1, R) = \{(q_1, GR)\} \\
\delta(q_1, 0, B) = \{(q_1, BB)\} & \delta(q_1, 1, B) = \{(q_1, GB)\} \\
\delta(q_1, 0, G) = \{(q_1, BG)\} & \delta(q_1, 1, G) = \{(q_1, GG)\} \\
\delta(q_1, c, R) = \{(q_2, R)\} & \delta(q_1, c, B) = \{(q_2, B)\} \\
\delta(q_1, c, G) = \{(q_2, G)\} & \delta(q_2, 0, B) = \{(q_2, \varepsilon)\} \\
\delta(q_2, 1, G) = \{(q_2, \varepsilon)\} & \delta(q_2, \varepsilon, R) = \{(q_2, \varepsilon)\}
\end{array}$$

La interpretación es que si el autómata est en el estado q_1 y lee un 0 entonces permanece en el mismo estado y añade una B a la pila; si lo que lee es un 1, entonces añade una G; si lee una c pasa a q_2. En q_2 se saca una B por cada 0, y una G por cada 1.

Se llama descripción instantá nea o configuración de un autómata con pila a una tripleta

$$(q,u,\alpha) \in Q \times A^* \times B^*$$

en la que q es el estado en el se encuentra el autómata, u es la parte de la cadena de entrada que queda por leer y α el contenido de la pila (el primer símbolo es el tope de la pila).

Definición 44 *Se dice que de la configuración $(q,au,Z\alpha)$ se puede llegar a la configuración $(p,u,\beta\alpha)$ y se escribe $(q,au,Z\alpha) \vdash (p,u,\beta\alpha)$ si y solo si*

$$(p,\beta) \in \delta(q,a,Z)$$

donde a puede ser cualquier símbolo de entrada o la cadena vacía.

Definición 45 *Si C_1 y C_2 son dos configuraciones, se dice que se puede llegar de C_1 a C_2 mediante una sucesión de pasos de cá lculo y se escribe $C_1 \overset{*}{\vdash} C_2$ si y solo si exite una sucesión de configuraciones $T_1,\ldots,T_n$ tales que*

$$C_1 = T_1 \vdash T \vdash T_{n-1} \vdash T_n = C_2$$

Definición 46 *Si M es un APND y $u \in A^*$, se llama configuración inicial correspondiente a esta entrada, u, a (q_0,u,Z_0) donde q_0 es el estado inicial y Z_0 el símbolo inicial de la pila.*

Ejemplo 65 *En el caso del autómata con pila del ejemplo anterior tenemos*

$$(q_1,011c110,R) \vdash (q_1,11c110,BR) \vdash (q_1,1c110,GBR) \vdash (q_1,c110,GGBR) \vdash$$

$$(q_2,110,GGBR) \vdash (q_2,10,GBR) \vdash (q_2,0,BR) \vdash (q_2,\varepsilon,R) \vdash (q_2,\varepsilon,\varepsilon)$$

5.1.1. Lenguaje aceptado por un autómata con pila

Existen dos criterios para determinar el lenguaje aceptado por un APND:

a) Lenguaje aceptado por estados finales:

$$L(M) = \{w \in A^* : (q_0,w,Z_0) \overset{*}{\vdash} (p,\varepsilon,\gamma), p \in F, \gamma \in B^*\}$$

b) Lenguaje aceptado por pila vacía:

$$N(M) = \{w \in A^* : (q_0, w, Z_0) \overset{*}{\vdash} (p, \varepsilon, \varepsilon), p \in Q\}$$

En el primer caso, una palabra es aceptada, si se puede llegar a un estado final despues de consumir la entrada. En el segundo criterio, los estados finales no tienen ningún significado, y una palabra se acepta si cuando se termina de leer la entrada la pila se queda vacía.

Ejemplo 66 *En el caso del ejemplo 64, la palabra* $011c110$ *es aceptada por el autómata por el criterio de pila de vacía. Es decir,* $011c110 \in N(M)$. *De hecho, el lenguaje aceptado por el este autómata, según el criterio de la pila vacía es* $N(M) = \{wcw : w \in \{0,1\}^*\}$. *Para este mismo autómata* $L(M) = \emptyset$, *ya que al no haber estados finales, nunca se puede verificar la condición de aceptación. Una pequeña modificación del APND, lo tranformaría en una autómata que aceptase el mismo lenguaje, pero ahora por el criterio de estados finales. Solo hay que considerar* $M = (\{q_1, q_2, q_3\}, \{0, 1, c\}, \{R, B, G\}, \delta, q_1, R, \{q_3\})$ *donde*

$$\begin{array}{ll}
\delta(q_1, 0, R) = \{(q_1, BR)\} & \delta(q_1, 1, R) = \{(q_1, GR)\} \\
\delta(q_1, 0, B) = \{(q_1, BB)\} & \delta(q_1, 1, B) = \{(q_1, GB)\} \\
\delta(q_1, 0, G) = \{(q_1, BG)\} & \delta(q_1, 1, G) = \{(q_1, GG)\} \\
\delta(q_1, c, R) = \{(q_2, R)\} & \delta(q_1, c, B) = \{(q_2, B)\} \\
\delta(q_1, c, G) = \{(q_2, G)\} & \delta(q_2, 0, B) = \{(q_2, \varepsilon)\} \\
\delta(q_2, 1, G) = \{(q_2, \varepsilon)\} & \delta(q_2, \varepsilon, R) = \{(q_3, R)\}
\end{array}$$

Teorema 15 *a) Si M es un APND entonces existe otro autómata* M', *tal que* $N(M) = L(M')$

b) Si M es un APND entonces existe otro autómata M', *tal que* $L(M) = N(M')$.

Demostración.-

a) Si $M = (Q, A, B, \delta, q_0, Z_0, F)$, entonces el autómata M' se construye a partir de M siguiendo los siguientes pasos:

- Se añaden dos estados nuevos, q_0' y q_f. El estado inicial de M' será q_0' y q_f será estado final de M'.
- Se añade un nuevo símbolo a B: Z_0'. Este será el nuevo símbolo inicial de la pila.
- Se mantienen todas las transiciones de M, añadiéndose las siguientes:
  - $\delta(q_0', \varepsilon, Z_0') = \{(q_0, Z_0 Z_0')\}$
  - $\delta(q, \varepsilon, Z_0') = \{(q_f, Z_0')\}, \quad \forall q \in Q$

b) Si $M = (Q, A, B, \delta, q_0, Z_0, F)$, entonces el autómata M' se construye a partir de M siguiendo los siguientes pasos:

- Se añaden dos estados nuevos, q_0' y q_s. El estado incial de M' será q_0'.
- Se añade un nuevo símbolo a B: Z_0'. Este será el nuevo símbolo inicial de la pila.
- Se mantienen todas las transiciones de M, añadiéndose las siguientes:
  - $\delta(q_0', \varepsilon, Z_0') = \{(q_0, Z_0 Z_0')\}$
  - $\delta(q, \varepsilon, H) = \{(q_s, H)\}, \quad \forall q \in F, H \in B \cup \{Z_0'\}$
  - $\delta(q_s, \varepsilon, H) = \{(q_s, \varepsilon)\}, \quad \forall H \in B \cup \{Z_0'\}$

■

5.2. Autómatas con Pila y Lenguajes Libres de Contexto

Teorema 16 *Si un lenguaje es generado por una gramática libre del contexto, entonces es aceptado por un Autómata con Pila No-Determinístico.*

Demostración.- Supongamos que la gramática no acepta la palabra vacía. En caso de que acepte la palabra vacía se le eliminaría y despues se podría transformar el autómata para añadir la palabra vacía al lenguaje aceptado por el autómata.

Transformemos entonces la gramática a forma normal de Greibach. El autómata con pila correspondiente es $M = (\{q\}, T, V, \delta, q, S, \emptyset)$ donde la función de transición viene dada por

$$(q, \gamma) \in \delta(q, a, A) \Leftrightarrow A \to a\gamma \in P$$

Este autómata acepta por pila vacía el mismo lenguaje que genera la gramática. ■

Ejemplo 67 *Para la gramática en forma normal de Greibach:*

$$S \to aAA$$

$$A \to aS \mid bS \mid a$$

el autómata es

$$M = (\{q\}, \{a, b\}, \{A, S\}, \delta, q, S, \emptyset)$$

donde

$$\delta(q, a, S) = \{(q, AA)\}$$

$$\delta(q,a,A) = \{(q,S),(q,\varepsilon)\}$$

$$\delta(q,b,A) = \{(q,S)\}$$

Teorema 17 *Si $L = N(M)$ donde M es un APND, existe una gramática libre del contexto G, tal que $L(G) = L$.*

Demostración.-

Sea $M = (Q,A,B,\delta,q_0,Z_0,\emptyset)$, tal que $L = N(M)$. La gramática $G = (V,A,P,S)$ se construye de la siguiente forma:

- V será el conjunto de los objetos de la forma $[q,C,p]$, donde $p,q \in Q$ y $C \in B$, además de la variable S que será la variable inicial.
- P será el conjunto de las producciones de la forma
  1. $S \to [q_0,Z,q]$ para cada $q \in Q$.
  2. $[q,C,q_m] \to a[p,D_1,q_1][q_1,D_2,q_2]\ldots[q_{m-1},D_m,q_m]$
     donde $a \in A \cup \varepsilon$, y $C,D_1,\ldots,D_m \in B$ tales que

     $$(p,D_1D_2...D_m) \in \delta(q,a,C)$$

     (si $m = 0$, entonces la producción es $[q,A,p] \to a$).

Esta gramática genera precisamente el lenguaje $N(M)$. La idea de la demostración es que la generación de una palabra en esta gramática simula el funcionamiento del autómata no determinístico. En particular, se verificar que $[q,C,p]$ generar la palabra x si y solo si el autómata partiendo del estado q y llegando al estado p, puede leer la palabra x eliminando el símbolo C de la pila. ■

Ejemplo 68 *Si partimos del autómata $M = (\{q_0,q_1\},\{0,1\},\{X,Z\},\delta,q_0,Z_0,\emptyset)$, donde*

$$\begin{array}{ll} \delta(q_0,0,Z_0) = \{(q_0,XZ_0)\}, & \delta(q_1,1,X) = \{(q_1,\varepsilon)\} \\ \delta(q_0,0,X) = \{(q_0,XX)\}, & \delta(q_1,\varepsilon,X) = \{(q_1,\varepsilon)\} \\ \delta(q_0,1,X) = \{(q_1,\varepsilon)\}, & \delta(q_1,\varepsilon,Z_0) = \{(q_1,\varepsilon)\} \end{array}$$

las producciones de la gramática asociada son

$$
\begin{array}{l}
S \rightarrow [q_0, Z_0, q_0] \\
S \rightarrow [q_0, Z_0, q_1] \\
\\
[q_0, Z_0, q_0] \rightarrow 0[q_0, X, q_0][q_0, Z_0, q_0] \\
[q_0, Z_0, q_1] \rightarrow 0[q_0, X, q_0][q_0, Z_0, q_1] \\
[q_0, Z_0, q_0] \rightarrow 0[q_0, X, q_1][q_1, Z_0, q_0] \\
[q_0, Z_0, q_1] \rightarrow 0[q_0, X, q_1][q_1, Z_0, q_1] \\
\\
[q_0, X, q_0] \rightarrow 0[q_0, X, q_0][q_0, X, q_0] \\
[q_0, X, q_1] \rightarrow 0[q_0, X, q_0][q_0, X, q_1] \\
[q_0, X, q_0] \rightarrow 0[q_0, X, q_1][q_1, X, q_0] \\
[q_0, X, q_1] \rightarrow 0[q_0, X, q_1][q_1, X, q_1] \\
\\
[q_0, X, q_1] \rightarrow 1 \\
[q_1, X, q_1] \rightarrow 1 \\
[q_1, X, q_1] \rightarrow \varepsilon \\
[q_1, Z_0, q_1] \rightarrow \varepsilon
\end{array}
$$

Elimando símbolos y producciones inútiles queda

$$
\begin{array}{l}
S \rightarrow [q_0, Z_0, q_1] \\
[q_0, Z_0, q_1] \rightarrow 0[q_0, X, q_1][q_1, Z_0, q_1] \\
[q_0, X, q_1] \rightarrow 0[q_0, X, q_1][q_1, X, q_1] \\
[q_1, X, q_1] \rightarrow 1 \\
[q_1, X, q_1] \rightarrow \varepsilon \\
[q_1, Z_0, q_1] \rightarrow \varepsilon
\end{array}
$$

Después de esta equivalencia un lenguaje independiente del contexto puede definirse indistintamente como aquel que es generado por una gramática de tipo 2 o bien como aquel que es aceptado por un autómata con pila.

5.3. Lenguajes Independientes del Contexto Deterministas

Definición 47 *Un autómata con pila se dice que es* **determinista** *(APD) si y solo si se verifican las dos condiciones siguientes*

1. $\forall q \in Q,\ Z \in B,$ *si* $\delta(q, \varepsilon, Z) \neq \emptyset$ *entonces* $\delta(q, a, Z) = \emptyset, \quad \forall a \in A.$

2. $\forall a \in A \cup \varepsilon,\ z \in B,\ q \in F, \quad \delta(q, a, Z)$ *nunca contiene más de un elemento.*

El autómata del Ejemplo 64 es un autómata determinista.

La clase de lenguajes aceptados por un autómata con pila determinista es más reducida que la clase de los lenguajes aceptados por los autómatas con pila genéricos. Además, en el caso de los autómatas con pila deterministas, los dos criterios (pila vacía y estados finales) no son equivalentes.

Definición 48 *Un lenguaje L se dice que es* **independiente del contextos determinista** *si y solo si es aceptado por un autómata con pila determinista por el criterio de estados finales.*

Ejemplo 69 *El lenguaje aceptado por el autómata del Ejemplo 64 era* $L = \{wcw^{-1} : w \in \{0,1\}^*\}$. *El autómata era determinista, pero el criterio usado era el de estados finales. Sin embargo, el autómata se puede convertir en otro también determinista y que acepte L por el criterio de estados finales.*

Sea el autómata $M = (\{q_1,q_2\},\{0,1,c\},\{R,B,G\},\delta,q_1,R,\{q_3\})$ *donde*

$$\begin{array}{ll}
\delta(q_1,0,R) = \{(q_1,BR)\} & \delta(q_1,1,R) = \{(q_1,GR)\} \\
\delta(q_1,0,B) = \{(q_1,BB)\} & \delta(q_1,1,B) = \{(q_1,GB)\} \\
\delta(q_1,0,G) = \{(q_1,BG)\} & \delta(q_1,1,G) = \{(q_1,GG)\} \\
\delta(q_1,c,R) = \{(q_2,R)\} & \delta(q_1,c,B) = \{(q_2,B)\} \\
\delta(q_1,c,G) = \{(q_2,G)\} & \delta(q_2,0,B) = \{(q_2,\varepsilon)\} \\
\delta(q_2,1,G) = \{(q_2,\varepsilon)\} & \delta(q_2,\varepsilon,R) = \{(q_3,\varepsilon)\}
\end{array}$$

El autómata tiene un sólo estado final y lo que hemos hecho es que, al mismo tiempo que dejamos la pila vacía, pasamos a ese estado final.

Lo que hemos hecho en el ejemplo anterior es siempre posible y se puede demostrar el siguiente resultado.

Teorema 18 *Si L es aceptado por un autómata con pila determinista M por el criterio de pila vacía, entonces existe otro autómata con pila determinista* M' *que acepta el mismo lenguaje por el criterio de estados finales.*

Demostración.- La demostración es la misma que la del apartado a) del Teorema 15. Sólo hay que tener en cuenta que en la transformación que se realiza, si partimos de un autómata con pila determinista M, entonces el autómata con pila que se obtiene, M' es también determinista. ■

El resultado inverso no es cierto. Hay lenguajes que son aceptados por autómatas con pila por el criterio de estados finales, que no pueden ser aceptados por el criterio de pila vacía.

Ejemplo 70 *El siguiente autómata determinista acepta por el criterio de estados finales el lenguaje de las palabras del lenguaje sobre* $\{0,1\}$ *con la misma cantidad de ceros que de unos, donde* q_2 *es el estado final y* R *el símbolo inicial de la pila.*

$$\begin{array}{ll}
\delta(q_1,0,X)=\{(q_1,XX)\} & \delta(q_1,1,Y)=\{(q_1,YY)\} \\
\delta(q_1,1,X)=\{(q_1,\varepsilon)\} & \delta(q_1,0,Y)=\{(q_1,\varepsilon)\} \\
\delta(q_1,\varepsilon,R)=\{(q_2,R)\} & \delta(q_2,0,R)=\{(q_1,XR)\} \\
\delta(q_2,1,R)=\{(q_1,YR)\} &
\end{array}$$

Sin embargo, no puede ser aceptado por un autómata determinista por el criterio de pila vacía. El problema es el siguiente: si fuese posible, después de leer 0011 *se quedaría la pila vacía, pero entonces la palabra formada por* 001101 *no podría ser aceptada, porque al ser el autómata determinista, después de leer la primera parte de la parte,* 0011*, la pila se quedaría vacía y como un autómata no puede seguir funcionando con la pila vacía, la palabra completa* 001101 *no podría ser leida y, por tanto no es aceptada.*

El razonamiento del ejemplo anterior se puede generalizar a cualquier lenguaje en el que se de la misma situación: dos palabras u y v del lenguaje distintas tal que una es un prefijo de la otra (una subcadena del principio de la palabra). De hecho esta propiedad es fundamental para caracterizar a los lenguajes aceptados por un autómata con pila deterministas por el criterio de estados finales que lo son también por el criterio de pila vacía. En lo que sigue, formalizamos lo que que acabamos de decir.

Definición 49 *Un lenguaje* L *se dice que tiene la propiedad prefijo si no existen dos palabras* $u,v\in L$ *tales que* $u\neq v$ *y una es un prefijo de la otra.*

El siguiente teorema (que enunciamos sin demostración) establece la relación entre los dos criterios de aceptación en autómatas con pila deterministas.

Teorema 19 *Un lenguaje* L *puede ser acepatado por un autómata con pila determinista por el criterio de pila vacía si y solo si puede ser aceptado por un autómata con pila determinista por el criterio de estados finales y tiene la propiedad prefijo.*

Si tenemos un lenguaje, L, independiente del contexto determinista que no cumple la propiedad prefijo es fácil tranformarlo en otro lenguaje muy similar que cumple la propiedad prefijo y puede ser aceptado por un autómata con pila determinista por el criterio de pila vacía. Basta con considerar un símbolo $\$\notin L$ y entonces el lenguaje viene dado por

$$L\{\$\}=\{u\$:u\in L\}$$

Es como añadirle un símbolo de fin de palabra ($) a todas las palabras de L.

Finalmente, es importante que quede claro que hay lenguajes que son independientes del contexto, pero no deterministas. Como ejemplo, podemos considerar $L = \{u \in \{0,1\}^* : u = u^{-1}\}$. No vamos a demostrar formalmente que este lenguaje no es deterministas. Intuitivamente, las palabras de este lenguaje se reconocen metiendo símbolos en la pila hasta la mitad de la palabra (primera fase), y después de la mitad se sacan símbolos de la pila (segunda fase). Si aparece el símbolo inicial de la pila se pasa a un estado final y la palabra es aceptada. El problema es que cuando se está leyendo la palabra símbolo a símbolo no se sabe cuando estamos en la mitad de la palabra y cuando hay que pasar de meter símbolos a sacar. Si perminitimos que haya no determinismo esto se resuelve dando, en todo momento de la primera fase, la opción de cambiar de la primera a la segunda fase.

Capítulo 6

Propiedades de los Lenguajes Libres del Contexto

6.1. Lema de Bombeo

Comenzamos esta sección con un lema que nos da una condición necesaria que deben de cumplir todos los lenguajes libres de contexto. Nos sirve para demostrar que un lenguaje dado no es libre de contexto, comprobando que no cumple esta condición necesaria.

Lema 2 (Lema de Bombeo para lenguajes libres de contexto) *Sea L un lenguaje libre de contexto. Entonces, existe una constante n, que depende solo de L, tal que si $z \in L$ y $|z| \geq n$, z se puede escribir de la forma $z = uvwxy$ de forma que*

1. $|vx| \geq 1$
2. $|vwx| \leq n$, *y*
3. $\forall i \geq 0, uv^i wx^i y \in L$

Demostración.-
Sólo vamos a indicar una idea de cómo es la demostración. Supongamos que la gramática no tiene producciones nulas ni unitarias (si existieses siempre se podrían eliminar).

Supongamos un árbol de derivación de una palabra u generada por la gramática. Es fácil ver que si la longitud de u es suficientemente grande, en su árbol de derivación debe de existir un camino de longitud mayor que el número de variables. Sea N un número que garantice que se verifica esta propiedad. En dicho camino, al menos debe de haber una variable repetida. Supongamos que esta variable es A, y que la figura 6.1 representa el árbol de derivación y dos apariciones consecutivas de A.

■

Ejemplo 71 *Vamos a utilizar el lema de bombeo para probar que el lenguaje $L = \{a^i b^i c^i \mid i \geq 1\}$ no es libre de contexto.*

Supongamos que L fuese libre de contexto y sea n la constante especificada en el Lema de Bombeo. Consideremos la palabra $z = a^n b^n c^n \in L$, que tiene una longitud mayor que n. Consideremos que z se puede descomponer de la forma $z = uvxy$, verificando las condiciones del lema de bombeo.

Como $|vwx| \leq n$, no es posible para vx tener símbolos a y c al mismo tiempo: entre la última a y la primera c hay n símbolos. En estas condiciones se pueden dar los siguientes casos:

– $|vx|$ contiene solamente símbolos *a. En este caso para $i = 0$, $uv^0wx^0y = uwy$ debería de pertenecer a L por el lema de bombeo. Pero uwy contiene n símbolos b, n símbolos c, menos de n símbolos a, con lo que no podría pertenecer a L y se obtiene una contradicción.*

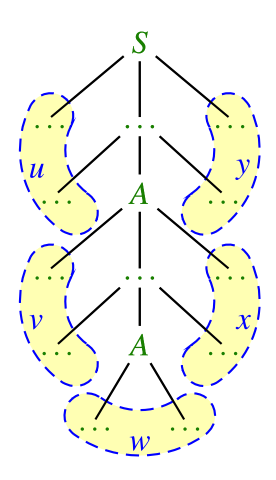


Figura 6.1: Arbol de Derivación en el Lema de Bombeo

- $|vx|$ contiene solamente símbolos *b. Se llega a una contradicción por un procedimiento similar al anterior.*

- $|vx|$ contiene solamente símbolos *c. Se llega a una contradicción por un procedimiento similar.*

- $|vx|$ contiene símbolos *a* y *b. En este caso, uwy tendría más símbolos c que a o b, con lo que se llegaría de nuevo a una contradicción.*

- $|vx|$ contiene símbolos *b* y *c. En este caso, uwy tendría más símbolos a que b o c, con lo que se llegaría también a una contradicción.*

En todo caso se llega a una contradicción y el lema de bombeo no puede cumplirse, con lo que L no puede ser libre de contexto.

Es importante señalar que el lema de bombeo no es una condición suficiente. Es solo necesaria. Así si un lenguaje verifica la condición del lema de bombeo no podemos garantizar que sea libre de contexto. Un ejemplo de uno de estos lenguajes es

$$L = \{a^i b^j c^k d^l | (i = 0) \vee (j = k = l)\}$$

Ejemplo 72 *Demostrar que el lenguaje* $L = \{a^i b^j c^i d^j : i, j \geq 0\}$ *no es libre de contexto.*

Ejemplo 73 *Demostrar que el lenguaje* $L = \{a^i b^j c^k : i \geq j \geq k \geq 0\}$ *no es libre de contexto.*

6.2. Propiedades de Clausura de los Lenguajes Libres de Contexto

Teorema 20 *Los lengaujes libres de contexto son cerrados para las operaciones:*

- *Unión*
- *Concatenación*
- *Clausura*

Demostración.-
Sean $G_1 = (V_1, T_1, P_1, S_1)$ y $G_2 = (V_2, T_2, P_2, S_2)$ dos gramáticas libres de contexto y L_1 y L_2 los lenguajes que generan. Supongamos que los conjuntos de variables son disjuntos. Demostraremos que los lenguajes $L_1 \cup L_2$, L_1L_2 y L_1^* son libres de contexto, encontrando gramáticas de tipo 2 que los generen.

- $L_1 \cup L_2$. Una gramática que genera este lenguaje es $G_3 = (V_1 \cup V_2 \cup \{S_3\}, T_1 \cup T_2, P_3, S_3)$, donde S_3 es una nueva variable, y $P_3 = P_1 \cup P_2$ más las producciones $S_3 \to S_1$ y $S_3 \to S_2$.
- L_1L_2. Una gramática que genera este lenguaje es $G_4 = (V_1 \cup V_2 \cup \{S_4\}, T_1 \cup T_2, P_4, S_4)$, donde S_4 es una nueva variable, y $P_4 = P_1 \cup P_2$ más la producción $S_4 \to S_1S_2$.
- L_1^*. Una gramática que genera este lenguaje es $G_5 = (V_1 \cup \{S_5\}, T_1, P_5, S_5)$, donde P_5 es P_1 más las producciones $S_5 \to S_1S_5$ y $S_5 \to \varepsilon$.

■

Algunas propiedades de clausura de los lenguajes regulares no se verifican en la clase de los lenguajes libres de contexto, como las que expresan el siguiente teorema y corolario.

Teorema 21 *La clase de los lenguajes libres de contexto no es cerrada para la intersección.*

Demostración.- Sabemos que el lenguaje $L = \{a^ib^ic^i | i \geq 1\}$ no es libre de contexto. Por otra parte los lenguajes $L_2 = \{a^ib^ic^j | i \geq 1 \text{ y } j \geq 1\}$ y $L_3 = \{a^ib^jc^j | i \geq 1 \text{ y } j \geq 1\}$ si lo son. El primero de ellos es generado por la gramática:

$$S \to AB$$

$$A \to aAb | ab$$

$$B \to cB | c$$

y el segundo, por la gramática:

$$S \rightarrow CD$$

$$C \rightarrow aC|a$$

$$D \rightarrow bDc|bc$$

Como $L_2 \cap L_3 = L_1$, se deduce que la clase de lenguajes libres de contexto no es cerrada para la intersección.

■

Corolario 1 *La clase de lenguajes libres de contexto no es cerrada para el complementario.*

Demostración.-
Es inmediato, ya que como la clase es cerrada para la unión, si lo fuese para el complementario, se podría demostrar, usando las leyes DeMorgan que lo es también para la intersección.

■

6.3. Algoritmos de Decisión para los Lenguajes Libres de Contexto

Existen una serie de problemas interesantes que se pueden resolver en la clase de los lenguajes libres de contexto. Por ejemplo, existen algoritmos que nos dicen si un Lenguaje Libre de Contexto (dado por una gramática de tipo 2 o un autómata con pila no determinístico) es vacío, finito o infinito. Sin embargo, en la clase de lenguajes libres de contexto comienzan a aparecer algunas propiedades indecidibles. A continuación, veremos algoritmos para las propiedades decidibles y mencionaremos algunas propiedades indecidibles importantes.

Teorema 22 *Existen algoritmos para determinar si un lenguaje libre de contexto es*

a) vacío

b) finito

c) infinito

Demostración.-

a) De hecho, ya hemos visto un algoritmo para determinar si el lenguaje generado por una gramática de tipo 2 es vacío. En el la primera parte del algoritmo para eliminar símbolos y producciones inútiles de una gramática, se determinaban las variables que podían generar una cadena fromada exclusivamente por símbolos terminales. El lenguaje generado es vacío si y solo si la variable inicial S es eliminada: no puede generar una palabra de símbolos terminales.

b) y c) Para determinar si el lenguaje generado por una gramática de tipo 2 es finito o infinito pasamos la gramática a forma normal de Chomsky, sin símbolos ni producciones inútiles. En estas condiciones todas las producciones son de la forma:

$$A \rightarrow BC,\ A \rightarrow a$$

Se construye entonces un grafo dirigido en el que los vértices son las variables y en el que para cada producción de la forma $A \rightarrow BC$ se consideran dos arcos: uno de A a B y otro de A a C. Se puede comprobar que el lenguaje generado es finito si y solo si el grafo construido de esta forma no tiene ciclos dirigidos.

■

Ejemplo 74 *Consideremos la gramática con producciones,*

$S \rightarrow AB$

$A \rightarrow BC|a$

$B \rightarrow CC|b$

$C \rightarrow a$

El grafo asociado es el de la figura 6.2. No tiene ciclos y el lenguaje es finito. Si añadimos la producción $C \rightarrow AB$, el grafo tiene ciclos (figura 6.3) y el lenguaje generado es infinito.

6.3.1. Algoritmos de Pertenencia

Estos algoritmos tratan de resolver el siguiente problema: dada una gramática de tipo 2, $G = (V,T,P,S)$ y una palabra $u \in T^*$, determinar si la palabra puede ser generada por la gramática. Esta propiedad es indecidible en la clase de lenguajes recursivamente enumerables, pero es posible encontrar algoritmos para la clase de lenguajes libres de contexto.

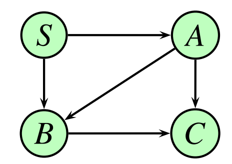


Figura 6.2: grafo asociado a una gramática de tipo 2 con lenguaje finito

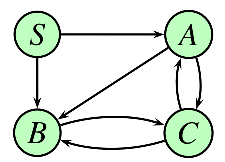


Figura 6.3: grafo asociado a una gramática de tipo 2 con lenguaje infinito

Un algoritmo simple, pero ineficiente se aplica a gramáticas en forma normal de Greibach (si una gramática no está en esta forma se pasa, teniendo en cuenta que hemos podido dejar de aceptar la palabra vacía). La pertencencia de una palabra no vacía se puede comprobar en esta forma normal de Greibach de la siguiente forma: Como cada producción añade un símbolo terminal a la palabra generada, sabemos que una palabra, u, de longitud $|u|$ ha de generarse en $|u|$ pasos. El algoritmo consistiría en enumerar todas las generaciones por la izquierda de longitud $|u|$, tales que los símbolos que se vayan generando coincidan con los de la palabra u, y comprobar si alguna de ellas llega a generar la palabra u en su totalidad. Este algoritmo para, ya que el número de derivaciones por la izquierda de una longitud dada es finito. Sin embargo puede ser muy ineficiente (exponencial en la longitud de la palabra). Para comprobar la pertenencia de la palabra vacía se puede seguir el siguiente procedimiento:

- Si no hay producciones nulas, la palabra vacía no pertence.

- Si hay producciones nulas, la palabra vacía pertenece si y solo si al aplicar el algoritmo que elimina las producciones nulas, en algún momento hay qhe eliminar la producción $S \rightarrow \varepsilon$.

El Algoritmo de Cocke-Younger-Kasami

Existen algoritmos de pertenencia con una complejidad $O(n^3)$, donde n es la longitud de la palabra de la que se quiere comprobar la pertenencia. Nosotros vamos a ver el algoritmo

de Cocke-Younger-Kasami (CYK). Este algoritmo se aplica a palabras en forma normal de Chomsky. Este consta de los siguientes pasos (n es la longitud de la palabra).

1. Para $i = 1$ hasta n
   2. Calcular $V_{i1} = \{A \mid A \rightarrow a$ es una produccion y el simbolo i-esimo de u es $a\}$
3. Para $j = 2$ hasta n
   4. Para $i = 1$ hasta $n - j + 1$
      5. $V_{ij} = \emptyset$
      6. Para $k = 1$ hasta $j - 1$
         $V_{ij} = V_{ij} \cup \{A \mid A \rightarrow BC$ es una produccion, $B \in V_{ik}$ y $C \in V_{i+k,j-k}\}$

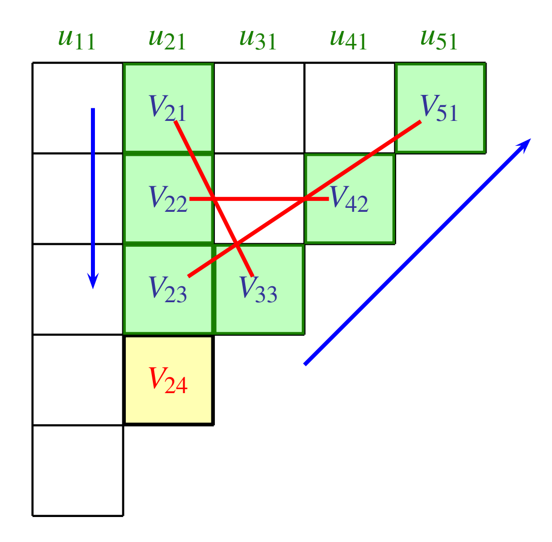
Este algoritmo calcula para todo $i, j (i \in \{1, \ldots, n\}, j \leq n - j + 1)$, el conjunto de variables V_{ij} que generan u_{ij}, donde u_{ij} es la subcadena de u que comienza en el símbolo que ocupa la posición i y que contiene j símbolos. La palabra u será generada por la gramática si la variable inicial S pertenece al conjunto V_{1n}.

Los cálculos se pueden organizar en una tabla como la de la siguiente figura (para una palabra de longitud 5):

u_{11}	u_{21}	u_{31}	u_{41}	u_{51}
V_{11}	V_{21}	V_{31}	V_{41}	V_{51}
V_{12}	V_{22}	V_{32}	V_{42}	
V_{13}	V_{23}	V_{33}		
V_{14}	V_{24}			
V_{15}				

En ella, cada $V_{i,j}$ se coloca en una casilla de la tabla. En la parte superior se ponen los símbolos de la palabra para la que queremos comprobar la pertenencia. La ventaja es que es fácil localizar los emparejamientos de variables que hay que comprobar para calcular cada conjunto V_{ij}. Se comienza en la casilla que ocupa la misma columna y está en la parte superior de la tabla, y la casilla que está en la esquina superior derecha, emparejando todas las variables de estas dos casillas. A continuación elegimos como primera casilla la que está justo debajo de la

primera anterior, y como segunda casilla la que ocupa la esquina superior derecha de la segunda anterior. Este proceso se continúa hasta que se eligen como primera casilla todas las que están encima de la que se está calculando. La siguiente figura ilustra el proceso que se sigue en las emparejamientos (para un elemento de la cuarta fila, en una palabra de longitud 5).



Ejemplo 75 *Consideremos la gramática libre de contexto dada por las producciones*

$S \rightarrow AB|BC$

$A \rightarrow BA|a$

$B \rightarrow CC|b$

$C \rightarrow AB|a$

Comprobar la pertenencia de las palabras $baaba, aaaaa, aaaaaa$ *al lenguaje generado por la gramática.*

El Algoritmo de Early

El algoritmo de Early es también de complejidad $O(n^3)$ en el caso general, pero es lineal para gramáticas $LR(1)$, que son las que habitualmente se emplean para especificar la sintaxis de las lenguajes de programación.

Al contrario que el algoritmo de Cocke-Younger-Kasami que trata de obtener las palabras de abajo hacia arriba (desde los símbolos terminales al símbolo inicial, el algoritmo de Early comenzará en el símbolo inicial (funciona de arriba hacia abajo).

Sea G una gramática con símbolo inicial S y que no tenga producciones nulas ni unitarias.

Supondremos que $u[i..j]$ es la subcadena de u que va de la posición i a la posición j. El algoritmo producirá registros de la forma (i, j, A, α, β), donde i y j son enteros y $A \to \alpha\beta$ es una producción de la gramática. La existencia de un registro indicará un hecho y un objetivo. El hecho es que $u[i+1..j]$ es derivable a partir de α y el objetivo es encontrar todos los k takes que β deriva a $u[j+1..k]$. Si encontramos uno de estos k sabemos que A deriva $u[i+1..k]$.

Para cada j, $REGISTROS[j]$ contendrá todos los registros existentes de la forma (i, j, A, α, β).

El algoritmos consta de los siguiente pasos:

P1 *Inicialización.*- Sea

- $REGISTROS[0] = \{(0,0,S,\varepsilon,\beta) : S \to \beta$ es una producción$\}$
- $REGISTROS[j] = \emptyset$ para $j = 1, \ldots, n$.
- $j = 0$

P2 *Clausura.*- Para cada registro $(i, j, A, \alpha, B\gamma)$ en $REGISTROS[j]$ y cada producción $B \to \delta$, crear el registro $(j, j, B, \varepsilon, \delta)$ e insertarlo en $REGISTROS[j]$. Repetir la operación recursivamente para los nuevos registros insertados.

P3 *Avance.*- Para cada registro $(i, j, A, \alpha, c\gamma)$ en $REGISTROS[j]$, donde c es un símbolo terminal que aparece en la posición $j+1$ de u, crear $(i, j+1, A, \alpha c, \gamma)$ e insertarlo en $REGISTROS[j+1]$.

Hacer $j = j+1$.

P4 *Terminación.*- Para cada par de registros de la forma $(i, j, A, \alpha, \varepsilon)$ en $REGISTROS[j]$ y $(h, i, B, \gamma A, \delta)$ en $REGISTROS[i]$, crear el nuevo registro $(h, j, B, \gamma A, \delta)$ e insertarlo en $REGISTROS[j]$.

P5 Si $j < n$ ir a P2.

P6 Si en $REGISTROS[n]$ hay un registro de la forma $(0, n, S, \alpha, \varepsilon)$, entonces u es generada. En caso contrario no es generada.

Ejemplo 76 *Comprobar mediante el algoritmo de Early si la palabra baa es generada por la gramática con producciones:*

$$\begin{array}{llll} S \to AB, & S \to BC, & A \to BA, & A \to a, \\ B \to CC, & B \to b, & C \to AB, & C \to a \end{array}$$

Después de aplicar el paso de inicialización, el contenido de $REGISTROS[0]$ es:

$REGISTROS[0] : (0,0,S,\varepsilon,AB), (0,0,S,\varepsilon,BC)$

$(0,0,A,\varepsilon,BA)$,
$(0,0,A,\varepsilon,a)$, $(0,0,B,\varepsilon,CC)$, $(0,0,B,\varepsilon,b)$, $(0,0,C,\varepsilon,AB)$, $(0,0,C,\varepsilon,a)$,
$REGISTROS[1]$:$(0,1,B,b,\varepsilon)$, $(0,1,S,B,C)$, $(0,1,A,B,A)$, $(1,1,C,\varepsilon,AB)$,
$(1,1,C,\varepsilon,a)$, $(1,1,A,\varepsilon,BA)$, $(1,1,A,\varepsilon,a)$, $(1,1,B,\varepsilon,CC)$, $(1,1,B,\varepsilon,b)$
$REGISTROS[2]$:$(1,2,C,a,\varepsilon)$, $(1,2,A,a,\varepsilon)$, $(0,2,S,BC,\varepsilon)$, $(0,2,A,BA,\varepsilon)$,
$(1,2,C,A,B)$, $(1,2,B,C,C)$, $(0,2,S,A,B)$, $(0,2,C,A,B)$, $(2,2,B,\varepsilon,CC)$,
$(2,2,B,\varepsilon,b)$, $(2,2,C,\varepsilon,AB)$, $(2,2,C,\varepsilon,a)$, $(2,2,A,\varepsilon,BA)$, $(2,2,A,\varepsilon,a)$
$REGISTROS[3]$:$(2,3,C,a,\varepsilon)$, $(2,3,A,a,\varepsilon)$, $(1,3,B,CC,\varepsilon)$, $(2,3,B,C,C)$
$(2,3,C,A,B)$, $(1,3,A,B,A)$

Como $(0,3,S,\alpha,\varepsilon)$ *no está en* $REGISTROS[3]$, *la palabra baa no es generada*

$S \to T$, $S \to S+T$, $T \to F$, $T \to T*F$, $F \to a$, $F \to b$, $F \to (S)$

Palabra: $(a+b)*a$

$REGISTROS[0]$:$(0,0,S,\varepsilon,T)$, $(0,0,S,\varepsilon,S+T)$, $(0,0,T,\varepsilon,F)$, $(0,0,T,\varepsilon,T*F)$, $(0,0,F,\varepsilon,a)$, $(0,0,F,\varepsilon,b)$, (
$REGISTROS[1]$:$(0,1,F,(,S))$, $(1,1,S,\varepsilon,T)$, $(1,1,S,\varepsilon,S+T)$, $(1,1,T,\varepsilon,F)$,
$(1,1,T,\varepsilon,T*F)$, $(1,1,F,\varepsilon,a)$, $(1,1,F,\varepsilon,b)$, $(1,1,F,\varepsilon,(S))$,
$REGISTROS[2]$:$(1,2,F,a,\varepsilon)$, $(1,2,T,F,\varepsilon)$, $(1,2,S,T,\varepsilon)$, $(0,2,F,(S,))$, $(1,2,S,S,+T)$
$REGISTROS[3]$:$(1,3,S,S+,T)$, $(3,3,T,\varepsilon,F)$, $(3,3,T,\varepsilon,T*F)$, $(3,3,F,\varepsilon,a)$, $(3,3,F,\varepsilon,b)$, $(3,3,F,\varepsilon,(S))$

$REGISTROS[4]$:$(3,4,F,b,\varepsilon)$, $(3,4,T,F,\varepsilon)$, $(1,4,S,S+T,\varepsilon)$, $(3,4,T,T,*F)$, $(0,4,F,(S,))$, $(1,4,S,S,+T)$
$REGISTROS[5]$:$(0,5,F,(S),\varepsilon)$, $(0,5,T,F,\varepsilon)$, $(0,5,S,T,\varepsilon)$, $(0,5,T,T,*F)$, $(0,5,S,S,+T)$,

$REGISTROS[6]$:$(0,6,T,T*,F)$, $(6,6,F,\varepsilon,a)$, $(6,6,F,\varepsilon,b)$, $(6,6,F,\varepsilon,(S))$,
$REGISTROS[7]$:$(6,7,F,a,\varepsilon)$, $(0,7,T,T*F,\varepsilon)$, $(0,7,S,T,\varepsilon)$, $(0,7,T,T,*F)$, $(0,7,S,S,+T)$

Como tenemos $(0,7,S,T,\varepsilon)$, *entonces la palabra* $(a+b)*c$ *es generada.*

6.3.2. Problemas Indecidibles para Lenguajes Libres de Contexto

Para terminar el apartado de algoritmos de decisión para gramáticas libres de contexto daremos algunos problemas que son indecidibles, es decir, no hay ningún algoritmo que los resuelva. En ellos se supone qhe G, G_1 y G_2 son gramáticas libres de contexto dadas y R es un lenguaje regular.

- Saber si $L(G_1) \cap L(G_2) = \emptyset$.
- Determinar si $L(G) = T^*$, donde T es el conjunto de símbolos terminales.
- Comprobar si $L(G_1) = L(G_2)$.
- Determinar si $L(G_1) \subseteq L(G_2)$.
- Determinar si $L(G_1) = R$.

- Comprobar si $L(G)$ es regular.
- Determinar si G es ambigua.
- Conocer si $L(G)$ es inherentemente ambiguo.
- Comprobar si $L(G)$ puede ser aceptado por una autómata determinístico con pila.

Bibliografía

[1] A.V. Aho, J.D. Ullman, *Foundations of Computer Science*. W.H. Freeman and Company, New York (1992).

[2] R.V. Book, F. Otto, *String rewriting systems*. Springer-Verlag, Nueva York (1993).

[3] J.G. Brookshear, *Teoría de la Computación. Lenguajes formales, autómatas y complejidad*. Addison Wesley Iberoamericana (1993).

[4] J. Carrol, D. Long, *Theory of Finite Automata with an Introduction to Formal Languages*. Prentice Hall (1989)

[5] D.I. Cohen, *Introduction to Computer Theory*. John Wiley, Nueva York (1991).

[6] M.D. Davis, E.J. Weyuker, *Computability, Complexity, and Languages*. Academic Press (1983)

[7] M.D. Davis, R. Sigal, E.J. Weyuker, *Computability, Complexity, and Languages, 2 Edic.*. Academic Press (1994)

[8] D. Grune, C.J. Ceriel, *Parsing techniques: a practical guide*. Ellis Horwood, Chichester (1990).

[9] M. Harrison, *Introduction to Formal Language Theory*. Addison-Wesley (1978)

[10] J.E. Hopcroft, R. Motwani, J.D. Ullman, *Introducción a la Teoría de Autómatas, Lenguajes y Programación,* 2ª Ed. Addison Wesley (2002).

[11] J.E. Hopcroft, J.D. Ullman, *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley (1979)

[12] J.M. Howie, *Automata and Languages.* Oxford University Press, Oxford (1991)

[13] H.R. Lewis, C.H. Papadimitriou, *Elements of the Theory of Computation*. Prentice Hall (1981)

[14] B.I. Plotkin, J.L. Greenglaz, A.A. Gvarami, *Algebraic structures in automata and database theory* World Scientific, River Edge (1992).

[15] G.E. Revesz, *Introduction to Formal Laguages*. Dover Publications, Nueva York (1991)

[16] T.A. Sudkamp, *Languages and Machines*. Addison Wesley, Reading (1988)